David Carlos Folgado da Cruz Piçarra

Licenciado em Engenharia Informática

Modelação de jogos colaborativos utilizando uma abordagem de grupos

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientadora : Carmen Pires Morgado, Profª. Auxiliar, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

Co-orientador : José Alberto Cardoso e Cunha, Prof. Catedrático, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

Júri:

Presidente: Prof. Doutor Carlos Augusto Isaac Piló Viegas Damásio

Arguente: Prof. Doutor Joaquim Belo Lopes Filipe

Vogal: Profª. Doutora Carmen Pires Morgado

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Maio, 2012

Modelação de jogos colaborativos utilizando uma abordagem de grupos

Agradecimentos

A dissertação que se segue e que, de resto, traduz o fechar de uma importante etapa na minha vida, obriga a um momento de reflexão e à conclusão de que este trabalho não constitui o esforço de uma só pessoa. Sinto, portanto, a necessidade de expressar a minha mais profunda gratidão por todos aqueles que têm contribuído para o meu sucesso.

Gostaria de agradecer, em primeiro lugar, aos meus orientadores, à Professora Carmen Morgado e ao Professor José Cardoso e Cunha, por me terem acompanhado neste desafio e pela disponibilidade demonstrada ao longo de todo o tempo de elaboração deste trabalho.

À minha família, que ultrapassa as fronteiras da consanguinidade e pela qual nutro o mais profundo orgulho e admiração: ao meu pai, Carlos Piçarra, pela confiança e apoio que cedo depositou em mim; à minha mãe, Lucília Folgado, pelo amor, força e coragem que sempre me transmitiu; ao meu irmão, João Duarte, pela constante cumplicidade e à Maria Fernanda Fontes, pela força e apoio incondicional.

Por último, não poderia faltar uma palavra de apreço ao punhado de amigos próximos de que alguém se pode rodear e aos quais tanto devo.

Resumo

Os modelos de grupos facilitam a organização de aplicações distribuídas. Por um lado, permitem identificar cada grupo como uma entidade, à qual se associa um nome e outros atributos comuns ao conjunto de membros que pertencem ao grupo. Por exemplo, os membros de um grupo podem cooperar para cumprir um objectivo comum ou podem aceder, de forma controlada, a recursos partilhados associados ao grupo. Por outro lado, os modelos de grupos facilitam a estruturação da comunicação entre os seus membros, na medida em que disponibilizam um espaço comum de interacção, no qual os membros podem comunicar entre si. Na maioria dos modelos de grupos, essa comunicação é efectuada através da difusão de mensagens (*multicast*).

Neste trabalho propõe-se uma plataforma, designada *Imagine*, que oferece uma interface de programação que se baseia num modelo de grupos desenvolvido em trabalhos anteriores (*MAGO - Modeling Applications with a Group Oriented approach*) e que integra múltiplos mecanismos de comunicação (por mensagens directas, por difusão de mensagens, por eventos assíncronos e por meio de um espaço de tuplos partilhado).

O objectivo principal deste trabalho foi o estudo da adequação do modelo de grupos acima mencionado, para suportar múltiplos tipos de interacção que se verificam em jogos colaborativos. Nesta dissertação descrevem-se as funcionalidades oferecidas pela plataforma *Imagine* e a arquitectura que suporta a sua implementação distribuída, com melhorias quanto à fiabilidade e eficácia, face à implementação anterior do modelo *MAGO*. Por forma a testar a plataforma desenvolvida e para validar as características do modelo de grupos mencionado, concebeu-se e implementou-se um jogo colaborativo.

Palavras-chave: Modelo de grupos, jogos colaborativos, espaço partilhado de memória.

Abstract

Group models are helpful to organize distributed applications. On one hand, group models can identify a group as an entity, assigning a name and other attributes common to the members that belong to the group. For example, group members can cooperate between themselves to achieve a common objective or access, in a controlled manner, shared resources dedicated to the group. On the other hand, group models facilitate the structuring of communication between group members, providing a common space of interaction, in which group members can communicate among each other. In most group models, this communication is achieved by multicast.

This work proposes a platform, named *Imagine*, which provides an application programming interface based on previously established group models (*MAGO - Modeling Applications with a Group Oriented approach*) and integration of several communication mechanisms (direct messages, multicast, events and shared space).

The main objective of this work was to study the adaptation of the mentioned group model, to support multiple interaction scenarios that occur in collaborative games. This thesis describes the features offered by the platform *Imagine* and the architecture that supports its distributed implementation, with improvements in the reliability and effectiveness of the previous implementation of the *MAGO* model. To test the developed platform and to validate the features of the group model previous mentioned, a collaborative game was conceived and implemented.

Keywords: Group models, colaborative games, shared memory space.

Conteúdo

Lista de Figuras

Lista de Tabelas

Listagens

1

Introdução

1.1 Motivação e Objectivos

Existe um crescente interesse no desenvolvimento de aplicações distribuídas que tirem partido das características dos modelos de grupos.

Esses modelos de grupos oferecem o seguinte conjunto de vantagens:

- Estruturação e organização dos participantes em grupos de membros cooperantes;
- Estruturação da interacção entre membros, no contexto de um espaço de interacção associado ao grupo. Na maioria dos modelos de grupos esta comunicação é suportada através de um modelo baseado em mensagens.
- Facilitar a expressão de comportamento dinâmico de aplicações, permitindo entradas e saídas dos membros dos grupos.

Nas últimas décadas, tem-se verificado um grande desenvolvimento em modelos de grupos, com diversas propostas de plataformas que suportam o conceito [Ban11] [Hal02] [BC01] [Cus08] [Mor07].

A maioria dos modelos de grupos, acima referenciados, são baseados em comunicação por mensagens e oferecem mecanismos de gestão de grupos e da sua filiação.

O trabalho apresentado nesta dissertação integra-se numa linha de desenvolvimento que visa estender as funcionalidades básicas oferecidas pelos modelos de grupos, com novos mecanismos de comunicação entre os membros de um grupo. Em particular, o modelo *MAGO* [Mor07], oferece uma interface de programação que permite aos membros de um grupo comunicarem através de mensagens (de forma directa ou por difusão), através de eventos e através de um espaço de tuplos partilhado.

O objectivo principal deste trabalho foi o estudo da adequação do modelo de grupos acima mencionado, com vista a facilitar o ensaio experimental de cenários de colaboração, em particular como os que se verificam em jogos colaborativos.

Os jogos colaborativos são um exemplo de aplicações distribuídas nos quais se podem utilizar grupos para melhorar as funcionalidades de desenvolvimento, pois estes permitem por um lado, a organização de jogadores em equipas para atingirem objectivos comuns e por outro lado permitem a comunicação entre jogadores. Para além disso, os jogos colaborativos apresentam características dinâmicas que podem ser capturadas através de modelos de grupos.

O recente desenvolvimento de aplicações baseadas na Internet, em particular jogos de múltiplos jogadores [Zyn11] [Ent11a] [EG11], torna cada vez mais relevante a necessidade de modelar os cenários de interacção e de colaboração que ocorrem em diferentes jogos.

Apesar do crescente desenvolvimento de jogos colaborativos e da disponibilização de plataformas de suporte ao seu desenvolvimento, verifica-se que a maioria destas apresentam limitadas funcionalidades para facilitar o ensaio experimental de cenários de colaboração, de forma flexível e de simples utilização.

Com esse objectivo, desenvolveu-se a plataforma *Imagine*, que disponibiliza uma interface de programação que suporta o modelo de grupos mencionado.

1.2 Trabalho Desenvolvido

O trabalho desenvolvido nesta dissertação dividiu-se em três grandes fases: reconhecimento das necessidades dos jogos colaborativos e mapeamento dessas necessidades sobre um modelo de grupos com espaço partilhado; concepção e implementação da plataforma *Imagine* que suporta o modelo; elaboração de um jogo colaborativo com o intuito de validação e ilustração da utilização da plataforma *Imagine*.

1.2.1 Funcionalidades requeridas para o desenvolvimento de jogos colaborativos

Na primeira fase da dissertação, avaliaram-se padrões de colaboração e interacção presentes em jogos de múltiplos jogadores, de forma a extrapolar as funcionalidades requeridas para o desenvolvimento destes. Como apresentado anteriormente, existem três funcionalidades importantes: interacção entre diversos jogadores; estruturação dos diferentes jogadores; dinamismo através da entrada e saída de jogadores durante um jogo.

Neste trabalho, pretendeu-se avaliar a adequação das características do modelo *MAGO* relativamente à gestão de grupos e da sua filiação, e às formas de comunicação suportadas - por mensagens, por eventos e por espaço partilhado, para suportar as funcionalidades presentes nos jogos colaborativos.

O mapeamento entre as funcionalidades requeridas pelos jogos e as características do modelo de grupos apresentadas considera as seguintes dimensões principais:

- A correspondência entre os mecanismos de comunicação disponíveis e as diferentes formas e semânticas de interacção entre jogadores;
- A forma como o conceito de grupos facilita a estruturação dos jogadores, quer de forma explícita, formando equipas, bem como de forma implícita, para confinar a interacção entre jogadores;
- A forma como o conceito de grupos permite capturar a natureza dinâmica de um jogo.

1.2.2 Concepção e implementação da plataforma *Imagine*

Por forma a suportar a experimentação dos mapeamentos referidos, desenvolveu-se a plataforma *Imagine* que suporta as seguintes funcionalidades:

- **Gestão de participantes**, responsável pela gestão de todos os participantes e respectiva informação associada a cada participante;
- **Gestão de grupos**, através da qual são suportadas todas as operações permitidas sobre grupos, isto é, criação e eliminação de grupos, bem como a gestão de filiação;
- **Comunicação**, permitindo as seguintes formas de interacção:
  - **Baseada em mensagens**, referente à troca de mensagens, quer directamente entre participantes, quer por difusão de mensagens entre participantes de um mesmo grupo;
  - **Baseada em eventos**, através de um modelo de publicação/subscrição;
  - **Baseada no conceito de espaço partilhado**, suportando acesso a um espaço de tuplos associado a cada grupo.

A plataforma desenvolvida disponibiliza as suas funcionalidades através de uma interface de programação em *Java*. As funcionalidades gráficas, por forma a tornar a interface visualmente mais interessante, são disponibilizadas através de um motor gráfico 2D (*JGame* [Sch11]), externo à plataforma.

A plataforma *Imagine* é suportada por uma arquitectura que assenta sobre os serviços oferecidos por *JXTA* e todas as funcionalidades suportadas encontram-se implementadas sobre a forma de um protótipo, tendo este protótipo sido utilizado no desenvolvimento do jogo colaborativo que serviu de teste e validação da plataforma *Imagine*.

1.2.3 Elaboração de um jogo colaborativo

De forma a validar as características do modelo de grupos e ilustrar a utilização da plataforma *Imagine*, procedeu-se à concepção e implementação de um jogo colaborativo sobre a plataforma.

O jogo desenvolvido é do tipo "caça ao tesouro" em que podemos ter diversas equipas a competir entre si. Cada equipa tem como objectivo coleccionar o maior número de tesouros presentes num campo de jogo no menor intervalo de tempo. A equipa vencedora é a que consegue capturar o maior número de tesouros no menor intervalo de tempo. A eficácia de cada equipa prende-se com a forma como a comunicação e a colaboração são estabelecidas entre os seus membros.

Este jogo tem como objectivo evidenciar as características oferecidas pela plataforma, no que diz respeito às funcionalidades requeridas pelos jogos de múltiplos jogadores. A interacção entre jogadores é efectuada ao longo do jogo, sendo possível aos membros de cada equipa comunicarem com um ou mais membros da sua equipa. Relativamente à estruturação dos jogadores, cada instância do jogo corresponde a um grupo que contém diversas equipas, sendo cada equipa representada por um grupo. Face ao dinamismo, no jogo que foi concebido é apenas possível aos participantes entrarem e saírem de grupos antes do jogo começar. No entanto, a plataforma permite efectuar essas entradas e saídas ao longo do jogo noutros cenários.

Por fim, foi efectuado o desenvolvimento deste jogo colaborativo sobre a plataforma *Imagine*, resultando deste desenvolvimento um protótipo do jogo.

1.3 Principais Contribuições

Esta dissertação apresenta três contribuições principais:

- Adaptação de um modelo de grupos com espaço partilhado para o desenvolvimento de jogos colaborativos;
- Concepção da plataforma *Imagine* baseada no modelo apresentado e implementação de um protótipo. A plataforma *Imagine* conta com os seguintes componentes: gestão de participantes; gestão de grupos; gestão de comunicação, sendo esta baseada em três formas de interacção: directa e difusão; por eventos; ou por espaço partilhado.
- Estruturação e desenvolvimento de um jogo colaborativo sobre a plataforma *Imagine*.

1.4 Estrutura do Documento

Neste primeiro capítulo foram apresentados os objectivos, motivações e principais contribuições desta dissertação.

No capítulo 2, é feito um levantamento das funcionalidades requeridas para o desenvolvimento de jogos colaborativos.

No capítulo 3, é apresentado um modelo de grupos e suas principais características, sendo apresentados dois exemplos de plataformas que implementam esse modelo de grupos. Seguidamente, discute-se a integração do conceito de espaço partilhado no contexto de modelos de grupos e o seu enquadramento em trabalhos anteriores e nos quais a plataforma *Imagine* se baseia.

No capítulo 4, descreve-se a plataforma *Imagine*, sendo neste capítulo descritos os objectivos e contribuições desta plataforma, bem como a sua arquitectura e as funcionalidades suportadas.

No capítulo 5, é descrita a concretização da plataforma *Imagine*, sendo apresentada uma discussão das principais opções de concepção. De seguida, descreve-se o mapeamento da arquitectura da plataforma *Imagine* sobre a plataforma *JXTA*.

No capítulo 6, é descrito um exemplo de um jogo colaborativo, servindo este jogo como validação e exemplo de utilização da plataforma *Imagine*.

No capítulo 7, são apresentadas possíveis extensões de efectuar sobre o jogo, sendo que a implementação das extensões é alcançada de forma rápida e simples, visto o jogo ter sido desenvolvido sobre a plataforma *Imagine*.

Por fim, no capítulo 8, são apresentadas as conclusões e são identificados possíveis trabalhos futuros.

2

Jogos Colaborativos

Este capítulo apresenta diferentes tipos de jogos de múltiplos jogadores, sendo analisados padrões de interacção presentes nesses jogos de forma a extrapolar as funcionalidades requeridas. Por fim, apresentam-se algumas plataformas de desenvolvimento de jogos e as funcionalidades disponibilizadas por estas.

2.1 Jogos de Múltiplos Jogadores

Com o crescimento exponencial da Internet, cada vez mais as pessoas estão ligadas entre si, facilitando assim o desenvolvimento de aplicações para diversos utilizadores, pelo que é natural que os jogos de múltiplos jogadores reflictam essa tendência.

Inicialmente, os jogos de múltiplos jogadores não permitiam ter jogadores diferentes em computadores ou plataformas de jogos separadas, como por exemplo, os jogos de arcada presentes em salões de jogos em que todos os diferentes jogadores estão ligados à mesma máquina e no mesmo espaço físico. Com a evolução dos protocolos de comunicação entre computadores, surge um crescente interesse em desenvolver jogos de múltiplos jogadores.

O primeiro jogo popular que tirou partido de um conjunto de protocolos de comunicação (em especifico os protocolos disponíveis no *AppleTalk* [Rev88]) que permitiu comunicação entre diferentes computadores numa rede local foi o *Spectre* [Com93]. O *Spectre* é um jogo em que na componente de apenas um jogador, o jogador conduz um tanque num mapa e tem como objectivo principal coleccionar bandeiras, encontrando pelo caminho munições para o seu tanque, bem como tanques inimigos controlados pelo próprio jogo.

No modo de múltiplos jogadores no jogo *Spectre*, cada jogador controla o seu tanque e

podem ser escolhidos três cenários possíveis, tendo cada cenário um objectivo diferente:

- **arena**, em que os jogadores ganham pontos ao destruir os tanques inimigos, que são controlados pelos restantes jogadores; o jogador vencedor é o jogador com mais pontos;
- **captura de bandeiras**, em que os jogadores têm como objectivo coleccionar bandeiras; o jogador que coleccionou mais bandeiras é o vencedor;
- **captura de bases**, em que os jogadores são divididos em duas equipas e cada equipa tem como objectivo capturar a base adversária; a equipa vencedora é a que conseguir capturar primeiro a base adversária.

2.1.1 Interacção entre os jogadores

Com a ligação das consolas recentes (Playstation 3 [Mic11] e Xbox 360 [Ent11b]) à Internet de forma constante, há interesse em promover a interacção entre os diferentes jogadores. Actualmente, a maioria dos jogos que se desenvolvem permitem a interacção com os restantes jogadores do mesmo jogo. Essa interacção com os jogadores pode ser efectuada de duas formas: directa ou indirecta.

2.1.1.1 Directa

A interacção directa entre os jogadores normalmente é alcançada através da troca de mensagens, tanto de texto como de voz.

Para facilitar a interacção directa, costuma-se organizar os jogadores que estão num mesmo jogo, sendo o caso mais habitual a inserção de diferentes jogadores numa mesma equipa e partilha de um mesmo objectivo. Por exemplo, no jogo *Dead Island* [Inc11], no qual o jogador tem como objectivo sobreviver numa ilha em que a maioria das pessoas foram infectadas e transformadas em *zombies*. Quando um jogador está a fazer uma missão e se detecta que existem mais jogadores a fazer a mesma missão, os jogadores são avisados da existência de outros jogadores na mesma missão e é possível formar uma equipa com esses jogadores.

2.1.1.2 Indirecta

Geralmente, a interacção indirecta assenta numa competição entre os diversos jogadores com vista à obtenção de maior ou melhor pontuação.

Este tipo de interacção costuma ocorrer através de pontuações globais ou de realizações pessoais que podem ser partilhadas com os restantes jogadores, geralmente referenciados, nos jogos para a Internet, como realizações pessoais (*achievements*). As realizações pessoais são objectivos secundários de um jogo, que não interferem com o desenrolar das histórias do jogo e que têm como objectivo aumentar a longevidade deste.

Relativamente a pontuações globais, no jogo *Angry Birds* [Ltd11] (que é um jogo no qual o jogador utiliza uma fisga para lançar pássaros em direcção a porcos que estão dentro de estruturas, com o intuito de destruir todos os porcos no mapa), após a conclusão de um determinado nível, é atribuída uma pontuação ao desempenho do jogador neste nível e é possível comparar essa pontuação com as dos restantes jogadores do mesmo jogo.

Quanto às realizações pessoais, no jogo *World of Warcraft* [Ent11a] (que é classificado como um *MMOG*, no qual o jogador é representado por uma personagem num mundo de fantasia e pode explorar esse mundo, lutar contra diversos monstros, fazer aventuras pré-definidas ou interagir com outros jogadores), existe um sistema de realizações associado a cada personagem que um jogador criou e é possível comparar essas realizações com qualquer personagem controlada por um jogador.

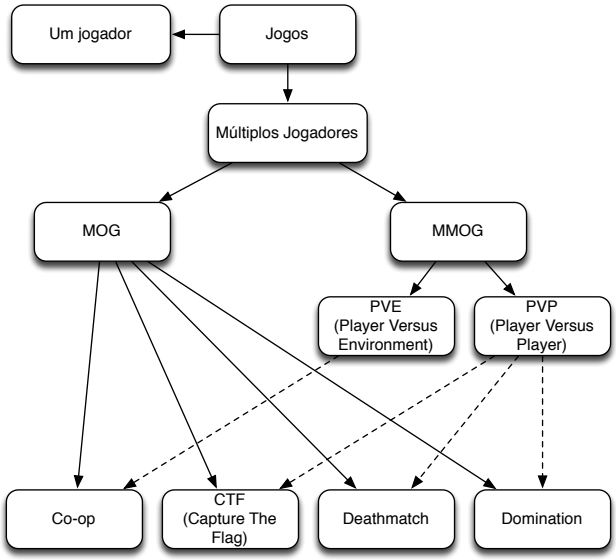

Figura 2.1: Diferentes tipos de jogos

2.1.2 Classificação dos tipos de jogos

Dentro dos diversos jogos de múltiplos jogadores é possível identificar os diferentes tipos de acordo com a persistência do cenário de jogo, ilustrado esquematicamente na figura 2.1:

- *Multiplayer online game* (*MOG*), este tipo de jogo é normalmente gerido por um servidor que está disponível na Internet. Esse servidor cria uma instância do jogo e permite que diversos jogadores possam interagir entre si. Após terminado o jogo essa instância é destruída. Os jogadores podem entrar e sair da instância a qualquer momento do jogo e geralmente, este tipo de jogos suporta até uma centena de jogadores numa mesma instância.

Um dos jogos mais conhecidos e populares deste género é o *Counter-strike* [Cor11], no qual os jogadores são divididos em duas equipas num mapa, os terroristas e os contra-terroristas. Os terroristas têm como objectivo colocar bombas em pontos estratégicos do mapa e os contra-terroristas têm como objectivo impedir esse atentado. Uma instância do jogo é dividida em rondas com um determinado tempo, no qual ganha a equipa que tiver mais sucesso no total dessas rondas. A interacção neste jogo é alcançada através da troca de mensagens entre os diferentes jogadores, sendo que numa instância é possível comunicar com todos os jogadores dessa instância ou comunicar apenas com os colegas de equipa.

Este jogo que até Agosto de 2011 vendeu mais de 25 milhões de unidades [Mak11], foi um dos principais responsáveis pela massificação dos jogos por rede, existindo equipas que são consideradas profissionais, e em que os jogadores recebem ordenados para jogar e são patrocinados por grandes empresas ligadas a este negócio, como por exemplo *Intel* ou *NVidia*.

- *Massively multiplayer online game* (*MMOG*), em contraste com o *MOG*, este tipo de jogo conta com uma instância persistente do mundo. Nessa instância, é possível que os jogadores entrem e saiam do jogo, sendo que a instância do mundo evolui independentemente desse dinamismo. Geralmente, este tipo de jogo suporta milhares de jogadores ao mesmo tempo.

  O jogo mais conhecido deste género é o *World of Warcraft*, que no final de Setembro de 2011 contava com 10.3 milhões de jogadores activos [Cif11]. Neste jogo é possível aos jogadores organizarem-se numa equipa e explorarem uma "masmorra"[1]. Cada equipa que entra numa masmorra conta com uma instância única dessa masmorra, de forma a que cada equipa possa derrotar e chegar aos seus tesouros, independentemente de outras equipas. Normalmente, uma masmorra permite apenas equipas de cinco jogadores mas existem masmorras que podem permitir até quarenta jogadores numa mesma instância.

  A interacção no *World of Warcraft* é efectuada através da troca de mensagens directas entre os diferentes jogadores, sendo também dada a possibilidade de interagir com as equipas que se formam ao longo do jogo.

  De forma a facilitar a procura de outros jogadores que contam com o mesmo interesse, isto é, completar uma determinada masmorra, existem grupos de jogadores que se organizam em equipas grandes (guildas). Essas guildas podem ter patrocínios, como o caso das equipas no jogo *Counter-strike*, no qual existe uma competição entre as diferentes guildas, face ao lançamento de novas masmorras, sendo que as primeiras guildas a completarem as novas masmorras servem de referência para as restantes guildas.

[1] Uma masmorra contém um conjunto de monstros e tesouros. Uma equipa pode explorar essa masmorra e derrotar os monstros de formar a chegar aos tesouros.

2.1.3 Ritmo do jogo

A maioria dos jogos de múltiplos jogadores são jogados em tempo real, isto é, as acções tomadas pelos jogadores são reflectidas no jogo e observadas, de forma imediata, pelos restantes jogadores.

Existem alguns jogos de múltiplos jogadores que são por turnos, sendo o exemplo mais conhecido destes os jogos *TBS* (*Turn-Based Strategy*). Um exemplo deste tipo de jogo é o *Civilization* [Sof11]. Este é um jogo do tipo *MOG* que permite que cada jogador conduza uma civilização desde a pré-história até ao futuro, sendo possível colocar diferentes objectivos finais para todos os jogadores, como por exemplo, conquista militar de todas as civilizações, primeira civilização a construir uma nave que permita viagens interestelares ou a civilização com a pontuação mais alta num determinado ano. Neste jogo um turno é definido por um conjunto de anos e em cada turno é possível mover as unidades do jogador, construir ou evoluir cidades, ou negociar com outras civilizações.

2.1.4 Competição nos jogos de múltiplos jogadores

Nos jogos de múltiplos jogadores existem diferentes formas de competição:

- Competição com o ambiente do jogo, como por exemplo, no jogo *World of Warcraft*, onde diversos jogadores se organizam em equipa para enfrentar desafios (masmorras) existentes no jogo.

- Competição entre jogadores, como por exemplo, no jogo *Quake* [LLC11] (em que cada jogador controla uma personagem numa instância de um mapa (*MOG*) e os mapas reflectem cenários futuristas no qual existem extraterrestres) é possível efectuar duelos entre dois jogadores;

- Competição entre equipas, como por exemplo, no jogo *Unreal Tournament* [EG11], que à semelhança do jogo *Quake*, permite que cada jogador controle uma personagem numa instância de um mapa (*MOG*) que representa um cenário futurista. Neste jogo formam-se equipas de jogadores e colocam-se as equipas numa mesma instância de um mapa a competirem por um determinado objectivo. É possível aos jogadores de uma instância comunicarem, através da troca de mensagens, com todos os jogadores dessa instância ou apenas com os jogadores da sua equipa;

Relativamente aos jogos do tipo *MMOG*, existe o conceito de *PVP* (*Player Versus Player*) e *PVE* (*Player Versus Environment*):

- *PVE* consiste em juntar diversos jogadores numa mesma equipa e oferecer desafios a essa equipa, como por exemplo as masmorras já previamente descritas.

- *PVP* consiste em incentivar os jogadores a competirem entre si. Normalmente são formadas equipas e incentiva-se a competição entre equipas. O número de jogadores por equipa depende do objectivo do jogo.

2.1.5 Modos de jogo

Independentemente do tipo de jogo, é possível identificar alguns modos de jogo, relacionados com o objectivo geral:

- *Co-op*: neste modo de jogo incentiva-se a colaboração entre os diversos jogadores para enfrentar desafios propostos pelo jogo, sendo esses desafios facilmente concluídos apenas se os jogadores se organizarem em equipa. A título de exemplo, no *World of Warcraft*, é possível a equipas de jogadores enfrentarem masmorras presentes no jogo.

- *CTF* (*Capture the flag*): este modo de jogo consiste em capturar o maior número de objectos, existindo bastantes jogos de múltiplos jogadores que têm este modo de jogo presente, como por exemplo o *Spectre*;

- *Deathmatch*: este modo de jogo consiste em colocar diversos jogadores num mesmo mapa e o vencedor desse jogo é o jogador que conseguir matar mais jogadores. O limite deste modo de jogo pode ser um número máximo de mortes ou um tempo limite. Existem diversos jogos que incorporam este modo de jogo, sendo os mais conhecidos o *Quake* e o *Unreal Tournament*;

- *Domination*: neste modo de jogo os jogadores têm como objectivo controlar pontos estratégicos presentes no jogo: normalmente, os jogadores são divididos em duas equipas e cada equipa tem como objectivo controlar o maior número de pontos estratégicos pelo maior tempo possível; como exemplo deste jogo tem-se o jogo *Unreal Tournament*.

Alguns jogos incorporam diferentes modos, como por exemplo o jogo *Unreal Tournament*, que conta com os modos de jogo *CTF*, *Deathmatch* e *Domination*.

2.2 Cenários de Colaboração em Jogos

Analisando os diferentes jogos apresentados na secção anterior, é possível identificar um conjunto de funcionalidades comuns requeridas para o desenvolvimento deste tipo de jogos.

2.2.1 Organização

Nos jogos de múltiplos jogadores, é fundamental que exista um mecanismo que permita identificar cada jogador presente no sistema, devendo também ser possível oferecer uma organização a esses jogadores, permitindo que estes tomem conhecimento uns dos outros e possam interagir entre si.

Normalmente, a constituição das equipas (grupos) é dinâmica e a sua criação depende do tipo de jogo:

- No caso dos *MOG*, as equipas estão pré-definidas e a filiação é efectuada ao longo do decorrer do jogo. Por exemplo, no jogo *Unreal Tournament* no modo de jogo *CTF*, é necessário que cada jogador conheça os membros da sua equipa, de forma a que os jogadores saibam quem são os seus colegas de equipa e os adversários. A filiação neste caso é efectuada de forma dinâmica, ou seja, quando um jogador entra nesta instância do jogo, tem de indicar qual é a equipa em que pretende filiar-se, ficando assim associado a essa equipa.

- No caso dos *MMOG*, a formação de equipas é efectuada ao longo do decorrer do jogo, por exemplo, no jogo *World of Warcraft*, é possível que diversos jogadores se organizem previamente, antes de entrar numa instância de uma masmorra. Se ao longo do decorrer da acção nessa masmorra, um jogador sair dessa equipa, o jogo tenta procurar um outro jogador para completar a equipa.

Em suma, normalmente as equipas são estabelecidas antes do jogo começar e é possível aos jogadores entrarem e saírem dessas equipas durante o decorrer do jogo.

2.2.2 Interacção

A interacção entre jogadores é um ponto fulcral nestes jogos, podendo a partilha de informação com os outros jogadores ser alcançada de diversas formas, que são descritas de seguida.

2.2.2.1 Baseada em mensagens

A troca de informação entre jogadores é efectuada através do envio de mensagens. Normalmente, as mensagens são entregues aos destinatários mal são enviadas, de forma a promover a colaboração em tempo real.

Geralmente, o envio de mensagens divide-se em dois aspectos:

1. *Ponto-a-ponto*: Uma das formas de interacção baseada em mensagens é a comunicação directa entre jogadores. Por exemplo, no jogo *Star Wars: The Old Republic*[2] [Bio11], um jogo do tipo *MMOG*, é possível comunicar directamente com outro jogador, sendo esta forma de comunicação comum para perceber qual o interesse de cada jogador e formar equipas com jogadores que partilhem o mesmo interesse.

2. *Difusão*: No contexto de uma equipa, deve ser possível aos jogadores partilharem informação com os restantes membros da sua equipa, sendo esta possibilidade alcançada através da difusão de mensagens. Continuando o exemplo anterior, no jogo *Star Wars: The Old Republic*, após formada uma equipa, é possível comunicar com todos os membros dessa equipa através da difusão de mensagens, sendo as mensagens recebidas por todos os membros.

[2]Este jogo é um concorrente do jogo *World of Warcraft*, sendo o universo deste jogo baseado no *Star Wars* e que conta com um conjunto de funcionalidades idênticas ao *World of Warcraft*.

2.2.2.2 Baseada em eventos

Por vezes, certas acções de jogadores podem alterar o estado do jogo e essa acção e respectiva alteração, devem ser reportadas de forma assíncrona aos restantes jogadores. Geralmente, estas alterações são reportadas à custa de eventos, sendo os eventos divulgados para os restantes jogadores presentes nessa instância do jogo.

Por exemplo, no jogo *Unreal Tournament*, quando se está no modo de jogo *Domination*, sempre que uma equipa captura um ponto estratégico, essa captura deve ser reportada para todos os jogadores dessa instância, de forma a que os jogadores adaptem as suas acções à alteração efectuada.

2.2.2.3 Baseada no conceito de espaço partilhado

Adicionalmente aos modos de comunicação apresentados, pode ser interessante partilhar informação através de um espaço comum a todos os jogadores, podendo esse espaço guardar informação de forma persistente. Por exemplo, no jogo *World of Warcraft*, é possível partilhar informação, de forma persistente, com jogadores de uma mesma guilda[3], através de pequenas mensagens.

2.2.3 Dinamismo

O dinamismo é um aspecto importante que torna os jogos imprevisíveis. Nos jogos de múltiplos jogadores, cada jogador altera o ambiente do jogo e não há maneira de saber ou prever o que os restantes jogadores vão fazer nesse ambiente.

Quanto mais dinâmico, aberto e com maior número de possibilidade de relações entre objectos, ou seja, quanto mais causalidade se verifica num jogo, mais dinâmico este se torna, pois dá a possibilidade aos jogadores de contribuírem de modos distintos para o ambiente do jogo [Par11].

A conjugação da imprevisibilidade e da causalidade dão a sensação, a cada jogador, de contribuir para o ambiente do jogo através da sua personagem ou das suas personagens.

É importante permitir o dinamismo (entrada e saída de jogadores) de forma a que os jogadores sintam a presença de outros jogadores, caso contrário, a sensação e experiência de contribuição externa são perdidas. As entradas e saídas de jogadores ao longo de um jogo permitem que a dificuldade desse jogo seja alterada dinamicamente, por exemplo, no *Quake* é possível a qualquer momento entrarem mais jogadores, desde que o servidor tenha capacidade para lidar com esse número de jogadores, e ao existirem mais jogadores no mesmo mapa, a dificuldade aumenta, dado que existem mais inimigos com comportamento imprevisível.

[3]Como referido anteriormente, uma guilda é uma equipa grande. Este termo é comum nos jogos do tipo *MMOG*.

2.2.4 Conclusão

Em suma, os cenários de colaboração presentes em jogos de múltiplos jogadores apresentam as seguintes características:

- *Organização*, segundo a qual as equipas são criadas antes do jogo começar ou durante o decorrer do mesmo, dependendo do tipo de jogo. A filiação a essas equipas é dinâmica, ou seja, a qualquer momento um jogador pode entrar ou sair de equipas.
- *Interacção* que ocorre de diversas formas, sendo os mecanismos de comunicação mais comuns baseados em mensagens, quer ponto-a-ponto bem como difusão, baseados em eventos ou baseados no conceito de espaço partilhado.
- *Dinamismo*, considerado fundamental em jogos de múltiplos jogadores, de forma a oferecer uma experiência única do jogo a cada jogador.

De seguida, apresenta-se um conjunto de plataformas de jogos que facilitam o desenvolvimento deste tipo de aplicações e suportam maioria das características apresentadas.

2.3 Plataformas de jogos

Existem diversas plataformas que permitem o desenvolvimento de jogos colaborativos. Para a utilização da maioria dessas plataformas, é em geral necessário obter uma licença comercial, não sendo disponibilizado o código fonte ou uma especificação mais técnica da plataforma, sendo dada um maior ênfase à elaboração do jogo (história) em si e da interface do jogo.

Todas as plataformas descritas nesta secção dão um grande ênfase à componente gráfica, sendo disponibilizado em todas as plataformas mencionadas, um motor gráfico 3D que permite o desenvolvimento de interfaces gráficas completas.

2.3.1 *Hero Engine*

A plataforma *Hero Engine* [Plc11b] é indicada para o desenvolvimento de jogos de grande escala (*MMOG*). Esta plataforma foi utilizada para desenvolver um recente jogo de grande escala, o jogo *Star Wars: The Old Republic* [Bio11].

Esta plataforma utiliza uma arquitectura cliente-servidor, na qual os jogadores são distribuídos por diferentes servidores de instâncias, associados a áreas do mapa e as informações importantes de cada jogador são guardadas num servidor global. A plataforma *Hero Engine* conta com os seguintes tipos de servidores (figura 2.2):

- *World*, este servidor é o único servidor que contém informação importante de todos os jogadores, como por exemplo quais as personagens de cada jogador e em que servidores de área se encontram;

- *Area*, que contém um conjunto de informação e processamento que podem representar uma porção do mapa;

- *Area Instance*, este servidor está sempre associado a um servidor do tipo *Area* e contém uma instanciação de uma porção do mapa, no qual os jogadores podem entrar nesta área e interagir com o jogo.

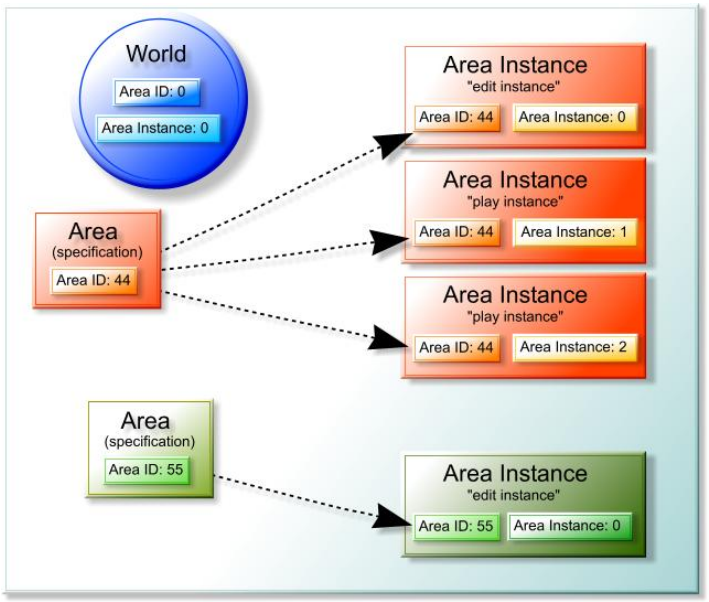

Figura 2.2: Diferentes servidores disponibilizados pela plataforma *Hero Engine* e a ligação entre os mesmos [Plc11a].

A programação efectuada nesta plataforma é baseada na linguagem de programação *HSL* (HeroScript Language), que é uma linguagem criada especificamente para o desenvolvimento de jogos nesta plataforma.

A linguagem de programação *HSL* tem uma sintaxe inspirada na linguagem de programação *Visual Basic*. Esta permite a programação orientada a objectos bem como programação por procedimentos, podendo o programador escolher o paradigma mais indicado para os cenários que pretende desenvolver.

Para programar sobre esta plataforma é necessário aprender uma linguagem de programação específica da plataforma, sendo também necessário programar e entender correctamente o propósito de cada servidor e a interacção entre estes, tornando assim a programação sobre esta plataforma uma tarefa algo complexa, com um longo período de adaptação aos paradigmas da plataforma.

Quanto à comunicação disponibilizada pela plataforma *Hero Engine*, existem dois mecanismos principais para comunicar com o servidor e para gerir a informação replicada: *RPC* (*Remote Procedure Call*) e *Replication*.

A comunicação baseada em *RPC* consiste na invocação de métodos remotos num determinado processo, no caso desta plataforma, o jogador consegue invocar métodos remotos num servidor.

O mecanismo *Replication* está associado a uma instância de uma zona e utiliza um mecanismo de replicação de informação entre os diversos jogadores de uma mesma instância.

Esta plataforma dá ênfase à integração e interacção de um jogador com o jogo, não existindo suporte explícito à organização de jogadores. Apesar da plataforma *Hero Engine* ser indicada para o desenvolvimento de jogos *MMOG*, a interacção entre jogadores é um ponto fulcral desses jogos e toda a interacção que se efectua entre os diversos jogadores é alcançada através de um conjunto de servidores disponibilizados para o jogo. Ao centralizar esta tarefa, adiciona-se carga adicional ao conjunto de servidores responsáveis pela gestão do jogo.

2.3.2 *Unity*

A plataforma *Unity* [Tec11b] permite o desenvolvimento de jogos para múltiplos jogadores bem como apenas para um jogador, sendo possível exportar esses jogos de forma a que possam ser executados em Windows, Mac, Xbox 360, PlayStation 3, Wii, iPad, iPhone e Android. Também é possível exportar o jogo como um *plugin* de forma a que possa ser executado a partir de um *browser*.

Relativamente aos jogos de múltiplos jogadores, esta plataforma apresenta uma arquitectura cliente-servidor, na qual existe um servidor principal que gere todos os jogadores e servidores que disponibilizam jogos.

Sempre que se pretende disponibilizar um jogo para múltiplos jogadores, é necessário registar o jogo num servidor principal de forma a que este seja conhecido pelo público, dado que não se comunica directamente com o servidor que disponibiliza o jogo mas sim com um servidor principal que serve de intermediário.

Existe um servidor principal público disponibilizado pela plataforma *Unity* por forma a facilitar a publicação de novos jogos, sendo no entanto possível criar o próprio servidor principal, de forma a ter um melhor controlo sobre as variáveis de controlo desse servidor.

A programação sobre a plataforma *Unity* é efectuada através de uma de três linguagens de programação: *UnityScript* [Uni11], uma linguagem proprietária baseada na especificação ECMA-262, que tem uma sintaxe idêntica à utilizada em *JavaScript*; *C#*; *Boo* [Tec11a], uma especificação de *Python*.

De forma a facilitar o desenvolvimento sobre esta plataforma, é incorporado o projecto *Mono* [Xam11] na plataforma. Este projecto tem o intuito de facilitar a exportação de um jogo para diversas plataformas.

Esta plataforma disponibiliza dois mecanismos de comunicação: *RPC* (*Remote Procedure Calls*) e *State Synchronization*.

A comunicação baseada em *RPC* é idêntica à disponibilizada pela plataforma *Hero Engine*, ou seja, permite invocar métodos remotos num determinado servidor.

O mecanismo de comunicação *State Synchronization* é utilizado para partilhar informação que está em constante actualização e divide-se em dois modos:

- *Reliable Delta Compressed* é um modo de transmissão de dados fiável, no qual é efectuada uma comparação dos últimos dados recebidos com os do cliente. Se não houve alteração nesses dados, então não é necessário transmitir de novo, caso contrário transferem-se os dados alterados.

- *Unreliable Delta Compressed* é um modo de transmissão de dados idêntico ao anterior mas não garante fiabilidade.

O melhor exemplo de utilização do mecanismo *State Synchronization* consiste na alteração da posição de um jogador, em que se difunde a posição do jogador pelos restantes jogadores presentes na rede.

A plataforma *Unity* enfatiza a integração de um jogador num jogo e todos os detalhes do jogo são centralizados, sendo necessário ter um conjunto de servidores que tenham capacidade para suportar o número pretendido de jogadores. Em contraste, poderiam distribuir-se algumas tarefas pelos jogadores, como por exemplo, a comunicação directa entre jogadores.

2.3.3 *Big World*

A plataforma *Big World* [Lim11a] é indicada para o desenvolvimento de jogos de grande escala (*MMOG*). Esta plataforma é particularmente relevante dado que, ao comprar-se uma licença comercial para utilizar esta plataforma, é também disponibilizado um conjunto de servidores para suportar os jogos desenvolvidos sobre a mesma.

À semelhança da plataforma *Hero Engine*, esta plataforma também suporta instanciação de áreas em servidores específicos[4]. Dado as licenças associadas a esta plataforma contarem com um conjunto de servidores para suportar os jogos desenvolvidos, existe um conjunto de servidores que se adapta consoante as necessidades de cada jogo.

Examinando a figura 2.3, verifica-se que o jogo *Game 1* é o jogo mais exigente de todos os jogos presentes na figura e conta com o maior número de servidores. Caso outro jogo comece a ter mais afluência, existe um mecanismo que adapta os servidores necessários aos respectivos jogos, dependendo das necessidades de cada jogo. Também é possível

[4]Na plataforma *Big World*, os servidores *Space* são idênticos aos servidores *Area* e os servidores *Shard* são idênticos aos servidores *Area Instance* presentes na plataforma *Hero Engine*.

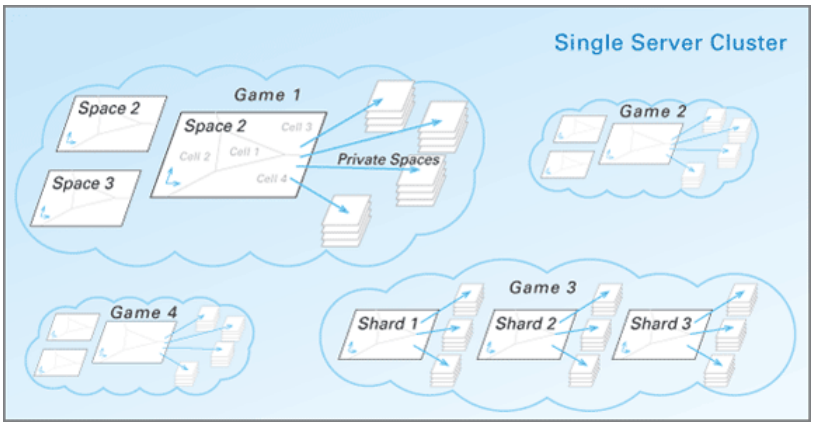

Figura 2.3: Exemplo de organização de um conjunto de servidores da plataforma *Big World* [Lim11b]

que os servidores afectados a um jogo sejam removidos, se se verificar que esses servidores se encontram sem carga que os justifique.

A programação sobre a plataforma *Big World* baseia-se na linguagem de programação *Python* e é possível utilizar a linguagem de programação *C++*, em casos específicos.

Esta plataforma também permite o desenvolvimento de aplicações simples para dispositivos móveis, sendo que essas aplicações permitem interagir com o jogo ou os jogadores presentes no jogo. Como exemplo, é possível desenvolver uma aplicação que permite que um utilizador comunique com os jogadores presentes no jogo, sem que o utilizador entre no jogo.

A plataforma *Big World* dá ênfase à ligação do jogador ao jogo, sendo que toda a interacção entre os diferentes jogadores é centralizada pelo conjunto de servidores disponibilizados pela empresa *Big World*.

2.3.4 *Prime Engine*

A plataforma *Prime Engine* [Pri11] foi uma das primeiras plataformas comerciais 3D e permite a construção de jogos do tipo *MMO*, que se executam a partir de um *browser*.

Esta plataforma utiliza uma arquitectura cliente-servidor, em que o cliente tem de descarregar um *plugin* de tamanho reduzido (menos de um megabyte) e executar esse *plugin* a partir de um *browser*. Para cada jogo podem existir um ou mais servidores que são responsáveis pela gestão do jogo e dos jogadores, sendo disponibilizada uma arquitectura escalável de servidores que suporta entre mil a dois mil utilizadores por servidor.

A programação nesta plataforma é baseada na linguagem de programação *Lua scripting* [RI11] e é efectuada através de um *toolkit* que permite a criação de instâncias de mundos persistentes de uma forma leve (*light persistent state worlds*).

A comunicação entre os jogadores e os servidores é baseada numa definição proprietária do protocolo *UDP* (*User Datagram Protocol*) que oferece fiabilidade e permite controlar a largura de banda, minimizar retransmissões e aumentar a rapidez na entrega de mensagens.

Esta plataforma tem como foco principal a integração do jogador no mundo de um jogo, no qual os servidores disponibilizados pelo jogo são responsáveis por toda a gestão do jogo, bem como pela comunicação entre os diferentes jogadores.

2.4 Conclusão

Este capítulo estudou os jogos de múltiplos jogadores e as suas diferentes características e analisou as funcionalidades requeridas no desenvolvimento desses jogos. Por fim, apresentou-se e analisou-se um conjunto de plataformas para o desenvolvimento de jogos.

Todas as plataformas analisadas contam com a seguinte característica: é dada ênfase à integração do jogador no jogo e toda a interacção entre os diferentes jogadores é alcançada através do conjunto de servidores disponibilizados para o jogo, não sendo oferecidos mecanismos explícitos de estruturação e organização entre jogadores.

No capítulo seguinte apresentam-se modelos e plataformas de suporte a grupos que dão ênfase à organização e interacção entre um conjunto de utilizadores.

3

Modelos e Plataformas de Suporte a Grupos

Este capítulo revê alguns modelos de comunicação existentes em plataformas de apoio ao desenvolvimento de aplicações distribuídas. Na primeira secção, são discutidos modelos de grupos e suas componentes principais. A segunda secção apresenta duas plataformas que implementam um modelo de grupos, sendo discutida a utilização dessas plataformas através da sua interface de programação. Por fim, é descrito o conceito de espaço partilhado e a sua integração com um modelo de grupos, sendo essa integração ilustrada nos casos de modelos e plataformas já existentes.

3.1 Modelo de Grupos

Dado um conjunto de processos que participem numa aplicação distribuída, os modelos de grupos surgem como método de organização com o intuito de facilitar a interacção entre diferentes processos, por forma a melhorar o desempenho, a fiabilidade ou a funcionalidade dessa aplicação.

A melhoria de desempenho verifica-se por exemplo quando existe um grupo de servidores, em que cada servidor consegue responder a pedidos. O modelo de grupos neste caso permite que os servidores consigam coordenar-se entre si, conseguindo assim dividir-se os pedidos pelos diferentes servidores presentes no grupo.

Quanto à melhoria de fiabilidade, esta geralmente está associada à tolerância às falhas. Esta melhoria verifica-se por exemplo quando ocorre uma falha num grupo de servidores, sendo possível que um dos servidores presentes nesse grupo mascare essa falha, substituindo o servidor que falhou.

Relativamente à melhoria da funcionalidade, esta ocorre através da utilização do conceito de grupo para modelar a colaboração e a cooperação entre um conjunto de participantes.

Um grupo consiste num conjunto de zero ou mais processos (chamados membros do grupo), no contexto do qual é disponibilizado um conjunto de mecanismos de comunicação[1] para facilitar a interacção entre os seus membros [Bir05].

Um grupo pode ter um coordenador, tendo este responsabilidade acrescida, como por exemplo, servir como mediador do processo de filiação de candidatos ao grupo ou verificação de falha de membros de um grupo através do envio de mensagens periódicas. O coordenador pode ser eleito de diversas formas. Por exemplo, o criador do grupo ou o membro mais antigo presente no grupo pode ser escolhido como coordenador.

A gestão de grupos é alcançada de diferentes formas, sendo analisadas mais à frente (na secção 3.2) duas plataformas que implementam esta gestão de maneiras distintas. Independentemente da gestão interna de grupos, deve ser possível ao programador criar ou destruir grupos no sistema.

3.1.1 Gestão de filiação

A gestão de filiação tem como objectivo gerir a constituição de um grupo, em particular lidando com as entradas e saídas dos seus membros. Essa gestão pode ser estática ou dinâmica [Bir05, p. 249-253]:

- A gestão estática consiste em definir previamente um conjunto de processos que se podem filiar no grupo, sendo esse conjunto estabelecido através de uma lista de identificadores ou nomes de membros que é definida aquando da criação do grupo.
- A gestão dinâmica permite que qualquer processo possa submeter um pedido de filiação ao grupo em qualquer momento.

Independentemente do tipo de gestão, pode existir um processo de validação de um candidato a membro relativamente a um conjunto de regras, como por exemplo, comparar dados fornecidos pelo candidato com dados já previamente conhecidos sobre esse candidato ou apenas aprovar um novo membro se houver um consenso numa votação realizada entre os membros presentes.

3.1.2 Gestão de comunicação

Normalmente, a comunicação num modelo de grupos é efectuada à custa de mensagens, sendo disponibilizados dois mecanismos de envio de mensagens: envio directo de mensagens entre dois participantes e difusão de mensagens pelos participantes de um grupo.

[1] Normalmente, a comunicação num modelo de grupos é baseada em troca de mensagens.

Relativamente à difusão de mensagens, esta tem duas abordagens possíveis: comunicação no contexto de grupos ditos fechados, no qual todas as mensagens trocadas dentro de um grupo são geradas por membros desse grupo; comunicação no contexto de grupos ditos abertos, em que pode haver mensagens enviadas para um grupo por parte de processos que não pertencem a esse grupo.

A comunicação num modelo de grupos tem dois aspectos fundamentais: fiabilidade e ordenação.

Relativamente à fiabilidade na comunicação, esta deve respeitar os seguintes princípios [CDK00]:

- **Validade**: se um participante difundir uma mensagem para todos os membros de um grupo, então é garantido que essa mensagem será entregue a todos os membros desse grupo;

- **Consenso**: se um membro de um grupo recebe uma mensagem m, então todos os membros desse grupo também recebem a mensagem m;

- **Integridade**: para todas as mensagens enviadas, cada membro irá entregar no máximo uma vez cada uma dessas mensagens.

Quanto à ordenação, os critérios de ordenação de mensagens mais frequentes [Bir05, p. 320-332] são os seguintes:

- *FIFO* (*First In First Out*): todas as mensagens emitidas pelo mesmo membro serão entregues pela ordem pela qual foram enviadas;

- Ordem causal: consiste em garantir que dada uma mensagem m que precede causalmente uma mensagem n, no sentido definido pela relação de precedência de Lamport[Lam78], então a mensagem m será entregue a todos os membros antes da mensagem n, mantendo assim a possível causalidade das mensagens, isto é, se a mensagem n surge em resposta à mensagem m, esta relação deve preservada na entrega das mensagens;

- Ordem total: consiste em garantir que todos os membros entregam as mensagens pela mesma ordem, ou seja, se uma mensagem m for entregue a um dos membros antes da mensagem n, então a mensagem m será entregue a todos os membros antes da mensagem n.

De forma a garantir estes dois aspectos, fiabilidade e ordenação, a entrega de uma mensagem divide-se em duas fases. Uma primeira fase no qual se recebe a mensagem e se aguarda até que esta mensagem esteja de acordo com a fiabilidade e ordenação estabelecidas entre os membros. Uma segunda fase tal que após verificada a fiabilidade e ordenação correcta da mensagem, se entrega a mesma à aplicação.

3.1.3 Eventos

Em aplicações para sistemas distribuídos é habitualmente necessário dispor de uma forma de comunicação associada a um mecanismo de notificação assíncrona de eventos. Do lado do emissor, permite-se assim reportar alterações dinâmicas ao estado do sistema. Do lado do receptor, o tratamento pode ser delegado num *handler*, executado como um *thread* cuja activação se efectua de forma assíncrona relativamente à execução do programa do receptor.

O modelo de comunicação baseado em eventos é assim, um modelo de comunicação importante num modelo de grupos.

O modelo de eventos mais comum assenta num mecanismo básico de publicação/-subscrição [Muh02] [EFGK03]. Este modelo permite que os emissores de eventos não necessitem de endereçar directamente os subscritores. Para os subscritores tomarem conhecimento desses eventos, é necessário registar (subscrever) o seu interesse na recepção de eventos de um determinado tipo.

3.1.4 Consistência do estado do grupo

As computações que resultam da execução de processos num sistema distribuído são definidas por conjuntos de eventos, sujeitos a uma certa ordenação. Cada evento representa uma transição no estado do sistema distribuído, podendo distinguir-se os eventos locais a um processo (por exemplo resultantes da execução de acções sobre as estruturas de dados locais e privadas ao contexto de execução do processo), dos eventos de interacção, que estão associados à comunicação entre processos (por exemplo, a emissão e recepção de mensagens, ou o acesso a estruturas de dados partilhadas).

Dada a natureza distribuída e assíncrona destes sistemas, cada um dos processos envolvidos numa computação distribuída constrói localmente uma observação da evolução do estado do sistema, sob a forma de uma sequência de eventos a qual é naturalmente diferente das observações construídas pelos outros processos.

Em muitas aplicações distribuídas, é necessário garantir que essas observações sejam consistentes, o que se traduz, por exemplo, na garantia de que todas as sequências observadas preservam a causalidade e/ou os eventos ocorrem nelas por uma mesma ordem, em todos os processos. No entando, nem todas as aplicações necessitam das mesmas garantias de consistência, pelo que as plataformas de suporte - incluindo as de programação baseadas em modelos de grupos - disponibilizam habitualmente, através das suas API de comunicação, diversas semânticas de consistência, para utilização mais conveniente pelas aplicações.

No caso dos modelos de grupos, as aplicações que envolvem a coordenação das acções dos membros dos grupos podem beneficiar de facilidades oferecidas pela plataforma de suporte, que lhe garantam, por exemplo, visões consistentes relativamente à constituição corrente de um dado grupo, ou seja a lista dos membros actuais. Como as acções de

entrada e saída de membros decorrem assincronamente, entre si, e relativamente ao envio e recepção de mensagens no grupo, um modelo de consistência que ofereça garantias relativamente à observação desses tipos de eventos, por todos os membros de um grupo, torna-se muito poderoso, facilitando a tarefa ao programador de aplicações distribuídas. Um exemplo significativo é do modelo de sincronia virtual.

O modelo de sincronia virtual permite que os programadores assumam que todos os membros de um grupo vêem os mesmos eventos e que esses eventos ocorrem pela mesma ordem em todas as visões. O modelo de sincronia virtual foi proposto pela primeira vez no sistema *Isis* [BJ87] [ISI11]. Este modelo garante a atomicidade dos eventos que ocorrem no sistema ou aplicação distribuída, independentemente do que ocorre internamente na entrega das mensagens a todos os membros. Por exemplo, se for detectada uma entrada de um membro enquanto se envia uma mensagem para os membros do grupo, esta entrada é registada na visão de cada membro de uma de duas formas: ou a entrada do membro é processada após a entrega da mensagem em todos os membros do grupo, sendo que este novo membro não recebe a mensagem; ou a entrada do membro é processada antes da entrega da mensagem e assim é incluído na entrega da mensagem.

Em termos formais, de forma a garantir a sincronia virtual e assumindo que uma visão representa a constituição do grupo, se um participante instala duas visões consecutivas v_i e v_{i+1} e entrega a mensagem m na visão v_i, então os restantes participantes que tenham as mesmas visões v_i e v_{i+1}, entregam m na visão v_i.

A desvantagem dos modelos como o de sincronia virtual é que, apesar das optimizações, implica sempre alguma penalização do desempenho de execução das aplicações, devido à necessidade de sincronização associada à garantia de consistência.

Por esse motivo e pelo facto de existirem diversas classes de aplicações que não necessitam de garantias de consistência tão apertadas, existem diversas plataformas de suporte que não as oferecem, através das suas API, ou apresentam diversas semânticas de consistência alternativas, à escolha do programador das aplicações.

3.2 Exemplos de Plataformas de Suporte a Grupos

Existem diversas plataformas que implementam o modelo de grupos, sendo nesta secção apresentados dois exemplos representativos.

O primeiro exemplo é a plataforma *JGroups* [Ban11]. Esta é relevante no universo de plataformas de suporte a grupos implementadas em *Java*, sendo um ponto de referência devido à sua grande utilização e aceitação por parte da comunidade. Esta plataforma oferece uma implementação robusta, na qual se fornece, ao nível da comunicação, uma pilha de protocolos flexível e é dada a possibilidade ao programador de configurar essa pilha de acordo com as necessidades do seu problema de aplicações.

O segundo exemplo é a plataforma *JXTA* [Hal02]. Esta é mais recente e mais leve do que *JGroups* (inclusivé, relativamente a uma abordagem mais "relaxada" face ao problema da consistência, atrás discutido), não permitindo uma configuração completa pelo

programador da pilha de protocolos. Esta plataforma permite a utilização de dispositivos móveis através da gestão dos endereços virtuais que identificam esses dispositivos. A plataforma *JXTA* é utilizada como base de implementação da plataforma *Imagine*, sendo os detalhes de implementação apresentados no capítulo 5.

3.2.1 *JGroups* - Sistema de comunicação fiável baseado em difusão de mensagens

A plataforma *JGroups* [Ban11] oferece um sistema de comunicação fiável baseado em difusão de mensagens. Esta plataforma oferece uma API disponível na linguagem de programação *Java*.

O sistema de comunicação fiável presente no *JGroups*, tem como ponto de partida o protocolo *IP multicast* e estende esse protocolo em dois aspectos: fiabilidade e modelo de grupos.

Um ponto forte desta plataforma é a personalização da pilha dos protocolos de comunicação, sendo possível escolher quais os protocolos que se pretende utilizar na aplicação que se está a desenvolver sobre esta plataforma, por exemplo, relativamente ao tratamento de falhas nas comunicações, à fragmentação de mensagens e à política de ordenação de mensagens, por exemplo, causal, total ou FIFO.

De seguida, apresentam-se algumas primitivas de comunicação e exemplos de utilização dessas primitivas.

3.2.1.1 *Channel*

A comunicação na plataforma *JGroups* baseia-se no conceito de *Channel* (canal).

Listing 3.1: Utilização de canais na plataforma *JGroups*

```
1 Channel c = new JChannel("stack.xml");
2 c.connect("Group");
3 Message msg = new Message(null, null, "Hello World");
4 c.send(msg);
5 msg = c.receive(0);
6 c.disconnect();
7 c.close();
```

Um processo pode definir vários canais. Cada canal tem um endereço único e uma definição da pilha de protocolos que o canal tem de respeitar. Para criar um canal, basta instanciar um objecto do tipo *JChannel*, passando como argumento a definição da pilha de protocolos, sendo esta definição efectuada através de um documento *XML* ou de uma *String*.

De forma a ilustrar a utilização de canais, apresenta-se a listagem de código 3.1 que contém um exemplo de utilização de canais. Após a criação de um canal (linha 1), é possível ligá-lo a um grupo, de forma a que este participante fique como membro desse grupo. Para ligar a um grupo, basta utilizar a primitiva *connect* (linha 2), passando como

argumento qual o nome do grupo a que este canal fica associado. Os grupos não necessitam de ser criados explicitamente: a criação dos mesmos ocorre quando um canal se tenta ligar a um grupo que ainda não existe.

O envio e recepção de mensagens são efectuados de forma explícita, existindo primitivas para o envio (*send*) e para a recepção de mensagens (*receive*).

De forma a enviar uma mensagem, basta instanciar um objecto do tipo *Message* (linha 3) e enviá-lo através da primitiva *send* (linha 4), sendo que esta instanciação do objecto do tipo *Message* recebe os seguintes argumentos:

- **receiver address**, indica qual o endereço para o qual se pretende enviar a mensagem; caso se pretenda enviar a mensagem para todos os membros desse grupo, basta submeter *null* neste argumento;
- **sender address**, indica qual o endereço do remetente; caso este argumento seja *null*, significa que a pilha de protocolos é responsável por preencher este campo com o valor correcto;
- **byte buffer**, indica o conteúdo da mensagem a enviar.

A recepção de uma mensagem é efectuada através da primitiva *receive* (linha 5), sendo a invocação da mesma bloqueante. Também é disponibilizada a primitiva *peek*, que permite a recepção de mensagens de forma não bloqueante.

Para sair de um grupo, é disponibilizada a primitiva *disconnect* (linha 6), que efectua a saída do grupo. Caso se pretenda fechar um canal, por exemplo porque o canal não vai ser mais utilizado ao longo do programa, é disponibilizada a primitiva *close* (linha 7).

A plataforma mantém a consistência do estado do grupo (suporta o modelo de sincronia virtual), notificando todos os membros quando existe alguma modificação na constituição do grupo.

3.2.1.2 *Building blocks*

Os canais presentes na plataforma *JGroups* apresentam primitivas de baixo nível e de acesso directo aos protocolos de comunicação. É possível definir *building blocks*, que oferecem uma abstracção de mais alto nível ao programador, e utilizam internamente os canais presentes na plataforma.

A título de exemplo, apresenta-se a listagem de código 3.2, que utiliza o *building block RcpDispatcher* disponibilizado pela plataforma. Essa classe possibilita a invocação de métodos remotos (*Remote Procedure Call*), de forma colectiva.

Listing 3.2: Utilização de *building blocks* na plataforma *JGroups*

```
1 public class RpcDispatcherTest {
2   public int print(int number) {
3     return number * 2;
4     }
5
```

```
6       public void start() throws Exception {
7       Channel c = new JChannel("stack.xml");
8       RpcDispatcher disp = new RpcDispatcher(c, null, null, this);
9       c.connect("Group");
10      RspList rsp_list = disp.callRemoteMethods(null, "print", new Integer(i),
            GroupRequest.GET_ALL, 0);
11      c.close();
12      }
13  }
```

A utilização do *building block* é alcançada através da instanciação de um objecto do tipo *RcpDispatcher* (linha 8). Quando se instancia esse objecto, é necessário indicar qual é o canal a que fica associado, sendo que esse canal será posteriormente ligado a um grupo.

Todas as invocações remotas são executadas através desse objecto, sendo disponibilizada a função *callRemoteMethods* (linha 10), onde é necessário indicar qual a função a executar e os parâmetros que se pretende passar na invocação. O resultado dessa função contém a invocação dessa função nos membros desse grupo.

A classe *RcpDispatcherTest* define uma função de exemplo *print* (linha 2 a 4), que recebe um número como argumento e devolve o dobro desse número. No exemplo apresentado, é invocada remotamente a função *print* em todos os membros do grupo *Group*.

3.2.1.3 Conclusão

A plataforma *JGroups* tem uma implementação robusta e oferece flexibilidade ao programador, pois é possível definir a configuração da pilha dos protocolos de comunicação. Esta plataforma é indicada para o desenvolvimento de aplicações complexas, nomeadamente que exijam semânticas de consistência, por exemplo, como o modelo de sincronia virtual.

No entanto, para os objectivos do trabalho que se apresenta nesta dissertação, e face às características das aplicações como os jogos colaborativos, torna-se mais adequada uma plataforma que suporte grupos de forma simples e flexível, inclusivamente relativamente ao suporte de diferentes dispositivos computacionais e à sua fácil integração num ambiente de execução.

3.2.2 *JXTA* - Conjunto de protocolos ponto-a-ponto

A plataforma *JXTA* [GBSK02] [Gon01], que se revê nesta secção, representa uma abordagem alternativa ao *JGroups*, no que se refere ao suporte de mecanismos de comunicação em grupo.

JXTA (*Juxtapose*) é um conjunto de seis protocolos que foram desenvolvidos para suportar a troca de mensagens e a colaboração entre diversos dispositivos, independentemente da topologia da rede física que os interliga.

A plataforma *JXTA* tem por base três camadas distintas, apresentadas esquematicamente na figura 3.1. A primeira camada, *JXTA Core*, representa o conjunto de primitivas

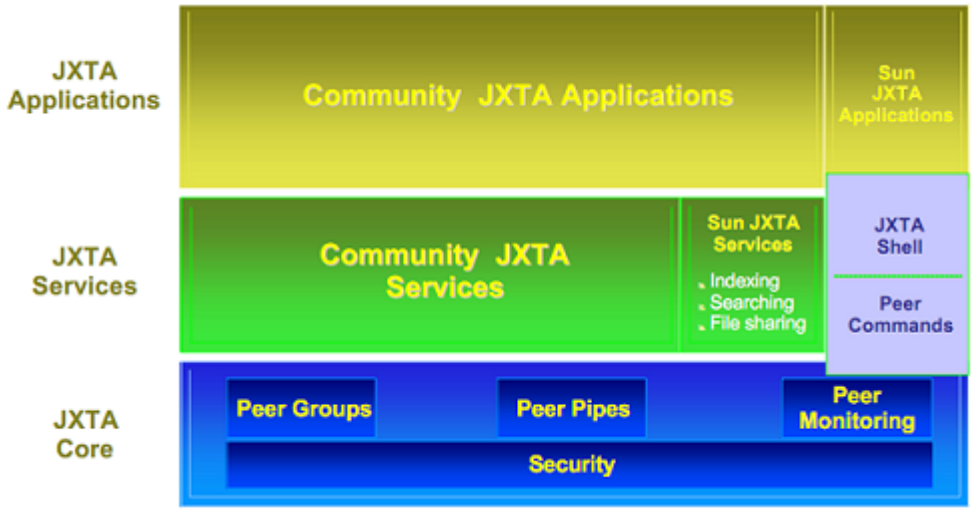

Figura 3.1: Arquitectura presente na plataforma *JXTA*, adaptado de [Hal02]

disponibilizadas pela plataforma. Essas primitivas dividem-se de acordo com os seis protocolos que formam esta plataforma: descoberta de participantes ou grupos (*Peer Discovery*; envio e recepção de consultas (*queries*) entre participantes (*Peer Resolver*); filiação de participantes a grupos(*Peer Membership*); criação e gestão de *pipes* (*Pipe Binding*); troca de mensagens directas (*Peer Endpoint*).

A segunda camada, *JXTA Services*, contém os serviços opcionais disponíveis para utilização da plataforma, sendo esta camada indicada para estender serviços disponibilizados pela primeira camada.

A terceira e última camada, *JXTA Applications*, suporta algumas operações comuns desenvolvidas que utilizam a plataforma *JXTA*, como por exemplo, partilha de ficheiros ou envio de mensagens, sendo disponibilizado acesso a essas operações através de uma interface de programação.

Dado que *JXTA* define seis protocolos e não numa implementação concreta da plataforma, é possível implementar *JXTA* em qualquer linguagem de programação. Nesta análise da plataforma *JXTA*, são descritas as funcionalidades e são mencionadas as que se encontram disponíveis na API em *Java*.

3.2.2.1 Participantes

Um participante (representado como um *Peer* em *JXTA*) é um ponto virtual de comunicação. Cada participante é representado na plataforma com um identificador único, independentemente do dispositivo onde está a ser executado, permitindo assim ter vários participantes sobre o mesmo dispositivo.

Cada participante na plataforma tem de estar registado, de forma a que todos os participantes sejam conhecidos na plataforma e possam comunicar entre si.

Na plataforma *JXTA* existem dois tipos de participantes:

- *Edge peer*: não será colocada responsabilidade extra nestes participantes, os quais poderão ter uma conexão limitada e apenas pretendem participar na plataforma como utilizadores básicos (em vez de serem fornecedores de serviços);
- *Super-peer*, divididos nos seguintes tipos:
  - *Rendezvous peer*: este participante fica encarregue de coordenar todos os participantes que tenham sido registados explicitamente neste *rendezvous*, garantido assim que todos os participantes registados irão receber todas as mensagens dirigidas a eles;
  - *Relay peer*: este participante possibilita que outros participantes consigam aceder à plataforma; por exemplo quando um participante não consegue utilizar a plataforma porque está por de trás de uma *firewall* que bloqueia os protocolos de comunicação utilizados na plataforma *JXTA*; esta possibilidade é conseguida através da retransmissão das mensagens através de outro protocolo, por exemplo *HTTP* (*Hypertext Transfer Protocol*).

Em termos da API, para que um participante entre na plataforma, é necessário que este efectue os seguintes passos (presentes na listagem de código 3.3).

Listing 3.3: Registar um participante na plataforma *JXTA*

```
1 JXTA.startNetwork(ADHOC, "User");
2 worldGroup = JXTA.worldGroup;
3 worldGroup.getRendezvousService().connectToRendezvous(addr);
```

Como ilustrado na listagem de código 3.3, os detalhes relativos a um participante, isto é, qual o modo em que este participante executa e qual o seu nome, são definidos na execução do método *JXTA.startNetwork* (linha 1), iniciando assim este participante na plataforma *JXTA* e criando um anúncio associado que identifica este novo participante (o mecanismo de anúncios é explicado de seguida na secção 3.2.2.3).

Existem os seguintes modos de execução:

- *ADHOC*: equivalente ao participante *Edge peer*;
- *Rendezvous*: equivalente ao *Rendezvous peer*;
- *Relay*: equivalente ao *Relay peer*;
- *Proxy*: este modo oferece reencaminhamento dos seis protocolos principais *JXTA* para dispositivos móveis baseados em *J2ME*;
- *Super*: este modo conjuga as funcionalidades dos modos *Rendezvous*, *Relay* e *Proxy*.

Após iniciada a plataforma *JXTA*, o participante efectua a filiação no grupo global. Este grupo global é importante, dado que os restantes grupos utilizam este como modelo, isto é, quais os serviços e funcionalidades de base que devem ter em cada grupo. O

grupo global também serve de ponto de referência para conhecimento de participantes e comunicação directa entre esses participantes.

A ligação a um *rendezvous peer* é efectuada explicitamente através do serviço de *RendezvousService*, indicando qual é o endereço deste participante.

3.2.2.2 Grupos

Um grupo na plataforma *JXTA* incorpora todas as características de um modelo de grupos e permite aos membros partilharem serviços específicos criados explicitamente pelo programador. Cada grupo disponibiliza os seis serviços base de *JXTA* que são apresentados na secção 3.2.2.5 e são acedidos através da interface de programação.

De forma a tornar um grupo privado é possível adicionar o conceito de filiação restringida, obrigando assim a que os participantes que pretendem entrar nesse grupo tenham de passar por um processo de autorização que valida a candidatura desse participante.

A filiação num grupo pode seguir dois modelos. No modelo de filiação local, em que a validação do candidato que se pretende juntar ao grupo é efectuada localmente no candidato[2], sendo possível, neste caso, juntar-se a um grupo sem comunicar com os restantes membros desse grupo. No modelo de filiação remota, o candidato necessita de comunicar com os membros do grupo, de forma a obter credenciais válidas.

A manipulação de um grupo é alcançada através da classe *PeerGroup*, sendo a partir desta classe possível aceder a todos os serviços disponíveis nesse grupo, por exemplo, aceder ao serviço de filiação (*MembershipService*) que permite que um participante se junte a um grupo (*apply* e *join*) ou saia de um grupo (*resign*).

3.2.2.3 Anúncios

Um anúncio (*advertisement*) é definido por um documento *XML*, que contém a descrição de uma mensagem, participante, grupo ou serviço. Essa descrição define detalhes importantes que permitem a um utilizador ligar-se ao objecto descrito no anúncio.

Listing 3.4: Criação de um grupo na plataforma *JXTA*

```
1 PeerGroupAdvertisement adv = AdvertisementFactory.newAdvertisement();
2 adv.setName("Group");
3 adv.setPeerGroupID(worldGroup.getPeerGroupID());
4 worldGroup.getDiscoveryService().publish(adv);
```

Por exemplo, como se pode verificar na listagem de código 3.4, quando um participante pretende criar um grupo, esse participante tem de criar um anúncio associado a esse grupo (linha 1, 2 e 3) e após criado esse anúncio, deve divulgá-lo de forma que os restantes participantes tenham conhecimento da existência deste grupo. A divulgação do anúncio é efectuada por difusão (*multicast*) (linha 4, através do serviço de descoberta

[2] Assumindo que este candidato tem conhecimento dos mecanismos de validação de entrada no grupo.

DiscoveryService), de tal modo que todos os participantes que estão na mesma rede local física que o emissor do anúncio irão receber este anúncio, o qual é enviado para o participante *rendezvous* associado ao participante que emitiu o anúncio. Os participantes com a propriedade *rendezvous* desempenham um papel importante na distribuição dos anúncios, dado que guardam os anúncios gerados por todos os participantes associados a este e facilitam a pesquisa de anúncios na plataforma, sendo possível resolver qualquer referência a um anúncio comunicando apenas com um conjunto de *rendezvous peers*.

A obtenção de anúncios por parte dos participantes é alcançada através do serviço de descoberta (*DiscoveryService*), que permite efectuar uma pesquisa por anúncios com uma determinada propriedade, como exemplificado na descrição deste serviço na secção 3.2.2.5.

3.2.2.4 Modelo de comunicação

O modelo de comunicação presente na plataforma *JXTA* (figura 3.2) é baseado no envio explícito de mensagens, as quais são recebidas assincronamente através de objectos ditos de escuta (*listeners*).

Tendo em conta que é necessário ter objectos de escuta para a recepção de mensagens, é necessário registar esses objectos de escuta do participante, num determinado grupo e associando um nome único e um parâmetro a cada um desses objectos de escuta. Esse registo é efectuado ao nível do serviço de *Endpoint*. O registo de objectos de escuta no grupo global possibilita que este participante receba mensagens directas, não associadas ao contexto de um grupo específico.

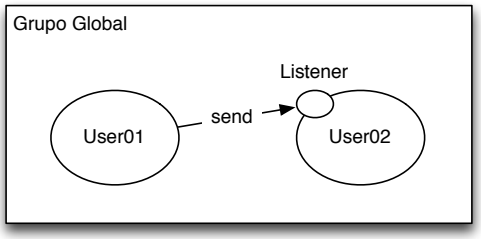

Figura 3.2: Exemplo de organização de participantes e seus listeners na plataforma *JXTA*

No envio de uma mensagem, é sempre necessário indicar qual o objecto de escuta que irá interpretar a mensagem enviada, sendo esse objecto de escuta associado a um grupo e identificado por um nome e parâmetro.

A utilização de um nome único e um parâmetro para identificar um objecto de escuta permite que seja possível agrupar objectos de escuta. Por exemplo, é possível registar diversos objectos de escuta com o nome "listener" mas com um parâmetro diferente, criando assim uma ligação lógica, para o programador, entre os diversos objectos de escuta registados.

A figura 3.3 ilustra os diferentes níveis de comunicação presentes na plataforma *JXTA*:

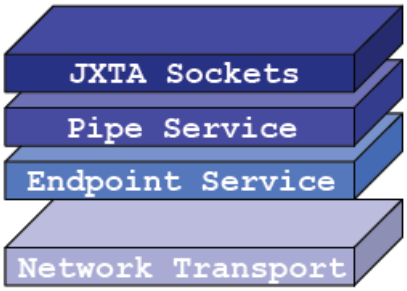

Figura 3.3: Pilha das componentes de comunicação na plataforma *JXTA*, adaptado de [Nob04]

endpoints são o nível mais simples de comunicação; *pipes* oferecem um canal de comunicação virtual entre participantes; *sockets*, nível de abstracção mais alto baseado na classe *Socket* presente em *Java*.

Comunicação por *Endpoints*

O *endpoint* é o protocolo de mais baixo nível existente no *JXTA* que permite que os participantes comuniquem directamente entre si.

Um participante pode ter diversos *endpoints* associados a si, possibilitando a utilização de diferentes protocolos de comunicação associados a cada *endpoint*. O endereço de um *endpoint* difere do endereço de *IP*: a separação destes dois conceitos premite associar o conceito de mobilidade ao participante, dado que o participante pode mudar de endereço de *IP* e os *endpoints* associados a esse participante mantêm-se inalterados.

A listagem de código 3.5 demonstra um exemplo de utilização de *endpoints*.

Listing 3.5: Utilização de *endpoints* na plataforma *JXTA*

```
1 PeerAdvertisement peer = worldGroup.getDiscoveryService().
      getRemoteAdvertisements(null, "PEER", "name", "User", 10, null).nextElement
      ();
2 EndpointAddress endpoint = new EndpointAddress(peer.getPeerID().toURI());
3 Messenger m = worldGroup.getEndpointService().getMessenger(endpoint);
4
5 Message msg = new Message();
6 msg.addMessageElement("Hello", "Hello World Message!");
7 m.sendMessage(msg, "hello", "");
```

O envio de uma mensagem para um participante, consiste em obter o anúncio relativo a esse participante (linha 1). Com o anúncio obtido, é criado um objecto *Messenger* (linha 2 e 3), que actua como mensageiro entre estes dois participantes e permite o envio de mensagens directas entre eles.

O envio de uma mensagem directa consiste na criação de um objecto do tipo *Message* (linha 5). Este objecto permite que o programador insira diversos pares de conteúdo.

Cada par é identificado por uma chave e o conteúdo dessa mensagem, no exemplo apresentado (linha 6), é inserido um par na mensagem com a chave *Hello* e conteúdo *Hello World Message!*.

De forma a enviar esta mensagem, basta invocar o método *sendMessage* (linha 7) associado ao *Messenger* previamente criado.

Listing 3.6: Recepção de uma mensagem na plataforma JXTA

```
1 class Listener extends EndpointListener {
2   public void processIncomingMessage(Message msg, EndpointAddress srcAddr,
        EndpointAddress dstAddr) {
3     // Do something...
4   }
5 }
6
7 worldGroup.getEndpointService().addIncomingListener(new Listener(), "hello", ""
    );
```

Como mencionado anteriormente, a recepção de mensagens é alcançada através de objectos de escuta, cada um dos quais associado a um nome, parâmetro e grupo, sendo assim necessário indicar no envio de uma mensagem, qual o objecto de escuta que deve interpretar a mensagem enviada. Quanto ao envio de mensagens directas, este está sempre associado ao grupo global e portanto não é necessário indicar no envio destas mensagens qual o grupo para que se pretende enviar a mensagem.

De forma a registar um objecto de escuta a um grupo basta aceder ao serviço de *endpoints* e invocar a primitiva *addIncomingListener*. Na listagem 3.6, é efectuado o registo de um objecto de escuta no contexto do grupo global (linha 7).

Comunicação por *Pipes*

Um *pipe* é uma ligação virtual entre participantes, que permite que os participantes comuniquem entre si.

Geralmente, a ligação entre dois participantes é vista como uma conexão directa, mas nem sempre se verifica esse tipo de conexão, ou seja, normalmente a ligação entre dois participantes envolve diversos dispositivos intermédios. Adicionalmente, podem existir participantes inacessíveis por alguns protocolos de comunicação, sendo necessário adaptar o canal de comunicação às restrições impostas pela rede que liga esses participantes.

Como abstracção destas dificuldades, surgem os *pipes* como uma camada que assenta sobre diferentes protocolos de comunicação e que suportam a propagação de mensagens à custa de outros participantes. Os *pipes* disponibilizados pela plataforma *JXTA* possuem um mecanismo de reconfiguração do *pipe* face a falhas da via de comunicação, sendo este mecanismo importante visto *JXTA* ser distribuído e dinâmico.

Existem diversos tipos de *pipes* definidos, destancando-se os actualmente implementados na versão disponível em *Java* [GBSK02]:

- *Unicast*: *pipe* mais básico presente na plataforma *JXTA*; é um canal ponto-a-ponto, assíncrono e não suporta segurança.

- *Unicast Secure*: este *pipe* é um canal ponto-a-ponto, assíncrono e que suporta segurança, através da codificação da informação transmitida com base no protocolo *TLS* (*Transport Layer Security*).

- *Propagate*: *pipe* que liga um emissor a diversos receptores, assíncrono e não suporta segurança.

A listagem de código 3.7, exemplifica a criação de um *pipe* e o envio de uma mensagem através desse *pipe*.

Listing 3.7: Utilização de *pipes* na plataforma *JXTA*

```
1 PipeAdvertisement pa = (PipeAdvertisement) AdvertisementFactory.
      newAdvertisement();
2 OutputPipe op = PeerGroup.getPipeService().createOutputPipe(pa, null);
3
4 Message msg = new Message();
5 msg.addMessageElement("Hello", "Hello World Message!");
6 op.send(msg);
```

Neste exemplo, é criado um anúncio que identifica um *pipe* genérico (linha 1). Após criado esse anúncio, cria-se um *pipe* com base nesse anúncio (linha 2). Por fim, envia-se uma mensagem idêntica à enviada na listagem de código 3.5 através da primitiva *send* (linha 7).

Comunicação por *Sockets*

Por fim, e como nível de abstracção mais alto no que toca à comunicação presente na plataforma *JXTA*, existe a noção de *sockets*. A comunicação baseada em *sockets* é orientada à emissão contínua de um fluxo de dados. A implementação deste conceito na versão implementada em *Java* é concretizada através da extensão da classe *Socket* presente em *Java* [Nob04].

3.2.2.5 Serviços

Como mencionado, *JXTA* é composto por seis protocolos, sendo a relação entre estes ilustrada na figura 3.4. Esses protocolos são disponibilizados na interface de programação como serviços. Segue-se uma breve descrição desses serviços.

Peer Discovery

Este serviço é essencial para a plataforma *JXTA*, descrevendo as operações possíveis quanto à descoberta de participantes ou grupos.

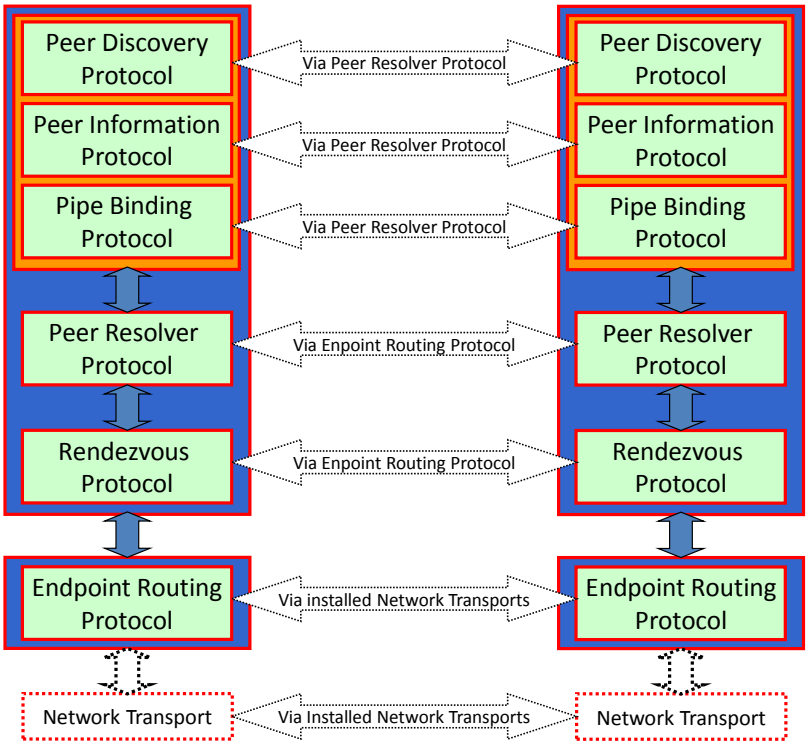

Figura 3.4: Relação existente entre os diferentes protocolos, adaptado de [Wil02]

Na implementação em *Java*, os anúncios que identificam participantes, grupos ou serviços e que são descobertos ao longo do decorrer da aplicação são guardados localmente. A plataforma *JXTA* utiliza um mecanismo de *cache* interna, para cada participante, no qual guarda todos os anúncios encontrados desde o início desse participante, sendo assim necessário disponibilizar dois tipos de descoberta. Na descoberta remota, este tipo de pesquisa utiliza o protocolo *peer resolver* para descobrir novos anúncios, sendo que todos os anúncios recebidos são guardados localmente. Na descoberta local, acede-se directamente aos anúncios que estão guardados localmente.

Listing 3.8: Serviço *Peer Discovery* na plataforma *JXTA*

```
1 PeerGroupAdvertisement pga = AdvertisementFactory.newAdvertisement();
2 PeerGroup.getDiscoveryService().publish(pga);
3 PeerGroup.getDiscoveryService().getRemoveAdvertisements(GROUP, "Group", null);
```

Relativamente à interface de programação, a listagem de código 3.8 ilustra a utilização deste serviço, no qual é criado um anúncio genérico a partir da "fábrica" de anúncios (*AdvertisementFactory*) (linha 1), sendo publicado esse anúncio (linha 2) e por fim, é efectuada uma pesquisa por todos os anúncios relativos a grupos que tenham o nome de grupo "Group" (linha 3).

Peer Resolver

Este serviço é utilizado para enviar uma consulta de um participante para outro participante e receber a resposta dessa consulta. Este serviço disponibiliza primitivas para o envio explícitos de consultas as quais são recebidas assincronamente através de *handlers*.

Todas as mensagens trocadas entre participantes são mensagens simples e sem garantias de entrega, quer no envio da consulta, bem como na recepção da resposta. Este serviço é utilizado internamente pelos restantes serviços para a divulgação de consultas.

Um exemplo da utilização deste serviço consiste na descoberta de novos participantes. Um participante ao iniciar-se na plataforma necessita de ter um anúncio associado a si e deve pública-lo na plataforma de forma a que os restantes participantes tenham conhecimento da sua existência. Essa publicação deve passar pela entrega do anúncio a pelo menos um participante com a propriedade de *rendezvous*. Quando se pretende obter informação quanto a um participante, é necessário efectuar uma consulta pelo anúncio associado a esse participante.

Listing 3.9: Serviço *Resolver* na plataforma *JXTA*

```
1 worldGroup.getResolverService().registerHandler("newparticipant", handler);
2 worldGroup.getResolverService().sendQuery(destPeer, query);
```

Quanto à interface de programação, a listagem de código 3.9 exemplifica a utilização deste serviço, em que se regista um *handler* (linha 1) que deriva da classe *QueryHandler* sobre o nome de *newparticipant* e em que se efectua o envio de uma consulta para um determinado participante da plataforma (*destPeer*) (linha 2). Este serviço encontra-se descrito em detalhe no livro *JXTA: Java P2P Programming* [GBSK02].

Peer Information

Este serviço é utilizado para saber o estado de um determinado participante. A propagação de informação relativa a um estado de um participante é alcançada através do envio explícito e da recepção assíncrona dessa informação.

Após obtido o anúncio relativo a um participante, é possível monitorizar o estado desse participante de forma a efectuar decisões adicionais quanto à utilização desse participante ou dos serviços disponibilizados por ele de forma eficaz.

Listing 3.10: Serviço *Peer Information* na plataforma *JXTA*

```
1 worldGroup.getPeerInfoService().getRemotePeerInfo(peer, monitorListener);
2
3 <?xml version="1.0" encoding="UTF-8"?>
4 <jxta:PeerInfoResponse xmlns:jxta="http://jxta.org">
5 <sourcePid> . . . </sourcePid>
6 <targetPid> . . . </targetPid>
7 <uptime> . . . </uptime>
8 <traffic>. . . </traffic>
9 </jxta:PeerInfoResponse>
```

A listagem de código 3.10 ilustra a utilização desse serviço, no qual se regista um *monitorListener* que irá monitorizar o estado de um determinado participante (linha 1). Esse *listener* irá receber respostas com a estrutura apresentada no resto da listagem de código (linha 3 a 11), que contém diversa informação útil relativa ao estado de um participante.

Este serviço encontra-se descrito em detalhe no livro *JXTA: Java P2P Programming* [GBSK02].

Peer Membership

Este serviço é responsável pela filiação de participantes a grupos.

É através deste serviço que é possível definir restrições quanto à filiação de novos participantes a um certo grupo, sendo por exemplo possível efectuar uma votação entre os participantes do grupo para validar a entrada do novo participante no grupo.

Após a entrada num grupo, esse participante recebe uma credencial que comprova que pertence ao grupo. Este serviço não guarda os participantes associados aos grupos e dado que a plataforma *JXTA* é distribuída, a credencial é essencial para validar a filiação de um participante a um dado grupo, retirando a necessidade da existência de um mecanismo externo para controlo dos participantes de cada grupo.

Listing 3.11: Serviço *Membership* na plataforma *JXTA*

```
1 auth = PeerGroup.getMembershipService().apply(new AuthenticationCredential());
2 PeerGroup.getMembershipService().join(auth);
```

A manipulação deste serviço em termos de programação, é alcançada através da classe *MembershipService*, exemplificada na listagem de código 3.11, em que se cria uma credencial genérica, através da instanciação de um objecto do tipo *AuthenticationCredential* e submete-se essa credencial ao serviço de filiação (linha 1). Após obtida a autorização para entrar no grupo, efectua-se a entrada nesse grupo submetendo a autorização obtida (linha 2).

Pipe Binding

Este serviço descreve como é possível aos participantes criarem *pipes* entre si e trocarem informação através desses *pipes*.

Um exemplo de uma capacidade nova que esses *pipes* oferecem é a possibilidade de atravessar *firewalls*. Este problema é bastante simples de se resolver quando se tem uma rede com poucos participantes ou se utiliza o modelo cliente-servidor, mas se for numa rede distribuída pura (como no modelo *peer-to-peer*) com um número elevado de participantes, este problema torna-se difícil de contornar. Neste caso, o protocolo oferece uma abstracção que colmata as dificuldades encontradas nos protocolos de comunicação normais.

A utilização de *pipes* é alcançada através da classe *PipeService* e é exemplificada na listagem de código 3.7, presente na secção 3.2.2.4.

Peer Endpoint

Este serviço é responsável por concretizar a troca de mensagens directa ou através de canais.

A utilização deste serviço é alcançada através da classe *EndpointService* que disponibiliza um conjunto de primitivas que corresponde à utilização dos *endpoints* definidos anteriormente na descrição da pilha de comunicação presente na plataforma *JXTA*.

Na secção 3.2.2.4 é exemplificada a utilização deste serviço na listagem de código 3.5.

3.2.2.6 Conclusão

JXTA é uma plataforma flexível que permite o desenvolvimento de aplicações distribuídas, sendo os detalhes de comunicação tratados pela plataforma. Esta plataforma é indicada para o desenvolvimento de aplicações simples dado que não obriga o programador a ficar dependente de semânticas de consistência fortes, disponibiliza diversos mecanismos flexíveis de comunicação, bem como permite a utilização de diferentes dispositivos computacionais e permite uma fácil integração num ambiente de execução.

3.3 Modelo de Grupos e Espaço Partilhado

Como referido no capítulo 1, a plataforma proposta e desenvolvida neste trabalho incorpora o conceito de espaço partilhado no contexto de um modelo de grupos.

O conceito de espaço partilhado permite que múltiplos processos distribuídos tenham acesso a estruturas de dados globais, ainda que a infraestrutura física não suporte memória partilhada pelo processadores reais que suportam a execução. Um espaço partilhado pode ser visto, em termos abstractos, como uma colecção de estruturas de dados, que podem ser manipuladas através de operações básicas de leitura (destrutiva ou não) e de escrita (*in-place* ou *out-of-place*). Estas operações podem ser invocadas concorrentemente por múltiplos processos, surgindo a necessidade de garantir a atomicidade das acções que manipulam as estruturas partilhadas e de definir a semântica de consistência que regula as operações de leitura, de escrita e a correspondente sincronização de processos (com invocações bloqueantes ou não).

O conceito pode ser implementado a níveis distintos de um sistema, desde o nível do sistema de operação ou o de *middleware* até ao nível de linguagens de programação. A nível do sistema operativo e/ou middleware, o conceito pode ser concretizado sob a forma de um espaço de "memória partilhada distribuída" (*Distributed Shared Memory*) [LH89]. A nível de modelos de programação, a proposta mais influente foi a do modelo Linda [CG89] [Gel85], por Carriero e Gelernter (que se revê na secção 3.3.1.1), que realiza o conceito através de um espaço de tuplos partilhados que permite a coordenação e sincronização de múltiplos processos distribuídos e cuja semântica de consistência associada às operações de leitura e escrita é muito simples, sem prejuízo da sua expressividade como modelo de coordenação.

O modelo Linda, ao concretizar o acesso às estruturas partilhadas num espaço de tuplos, separado dos processos distribuídos, permite uma forma de comunicação indirecta e anónima entre os processos, a qual é muito importante para modelar aplicações distribuídas em que os múltiplos emissores e receptores evoluem assincronamente e são criados/destruídos dinamicamente.

Para além disso, o espaço de tuplos pode ser visto como um repositório de informação, que facilmente pode ser estendido por forma a garantir o arquivo persistente dos tuplos. Devido às características apontadas, este modelo tem encontrado múltiplas realizações, desde os anos 80, seja a nível de API, de *middleware*, seja a nível de linguagem de programação [Bas95], incluindo *JavaSpace* [Ora11].

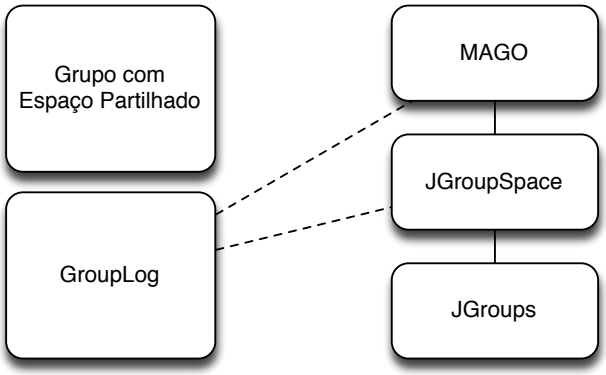

Figura 3.5: Ligação entre os diferentes modelos apresentados

O modelo *GroupLog* [BC01] [Bar04] oferece abstracções de programação para a organização de processos distribuídos com base em grupos. O modelo verifica as abstracções de comunicação entre processos, membros de um grupo, integrando formas de comunicação directa, colectiva e por meio de um espaço partilhado, com semântica inspirada no modelo *Linda*. Na sua especificação inicial, *GroupLog* foi definido com base numa linguagem de programação em lógica e foi implementado sobre uma camada intermédia de programação paralela baseada no sistema *PVM* [CM96]. Esta implementação, que visava a comprovar a validade dos conceitos do modelo, foi, contudo, baseada numa abordagem centralizada, assente num servidor global, responsável pela gestão dos grupos e pela sua coordenação.

No sentido de viabilizar a aplicação do módulo em aplicações reais, foi depois desenvolvido o modelo *JGroupSpace* [Cus08], com o objectivo de oferecer suporte, a nível de uma camada *middleware*, dos mecanismos básicos de gestão de grupos, de comunicação por mensagens, por eventos e por espaço de tuplos partilhados. A interface de programação de *JGroupSpace* está definida na linguagem *Java* e a implementação assenta na plataforma *JGroups* (anteriormente revista), tirando partido da semântica do modelo

de sincronia virtual, para a gestão da consistência e oferecendo uma implementação distribuída do espaço de tuplos.

O modelo *MAGO* [Mor07] teve como principal objectivo adaptar e estender as abstracções do modelo GroupLog, para aplicações interactivas baseadas em grupos, bem como viabilizar a sua utilização com sistemas de informação.

A figura 3.5 ilustra a relação entre os diferentes modelos apresentados nesta secção.

3.3.1 Modelo *GroupLog*

O modelo *GroupLog* [BC01] [Bar04] foi desenvolvido com base numa linguagem de programação em lógica e implementado com base no sistema *PVM* [CM96], baseado na linguagem de programação *Prolog*, dotado de extensões para concorrência e comunicação entre processos. O modelo está centrado no conceito de grupos, permitindo a organização de entidades, definindo mecanismos e abstracções que dêem suporte à coordenação das entidades. O modelo suporta entidades elementares (processos) e entidades compostas (grupos), permitindo a formação de hierarquias de grupos.

No modelo *GroupLog* os grupos, bem como as entidades que neles participam, são definidos por um nome, uma interface bem definida e um programa. Adicionalmente, cada grupo possui internamente um espaço partilhado, definido como um conjunto de tuplos. Esse espaço partilhado é utilizado para definir o estado do grupo, isto é, informação associada a um grupo que pode ser acedida e alterada por qualquer membro do grupo.

No modelo *GroupLog*, a interacção entre entidades, pertencentes a um mesmo grupo, é efectuada de duas formas:

- acesso ao espaço partilhado, cuja realização é baseada no modelo *Linda* [CG89] (que se revê na secção seguinte);

- comunicação directa através da invocação dos métodos declarados nas interfaces definidas pelas entidades. Cada interface associada à sua entidade descreve quais os métodos possíveis de executar nessa entidade; por exemplo, uma entidade pode ter definido na sua interface um método *copy(a,b)* que recebe um valor, o valor *a* que pode ser um valor qualquer e replica esse valor no segundo argumento.

Quanto ao espaço partilhado de tuplos, é de salientar que este tem as mesmas propriedades de um mediador de comunicação entre processos, ou seja, tem uma existência separada e independente das vidas dos processos, não exigindo que os processos comunicantes coexistam simultaneamente, sendo por isso ideal para realizar a coordenação entre processos concorrentes. Por outro lado, a forma de comunicação entre processos é anónima.

3.3.1.1 Modelo *Linda*

O modelo *Linda* [CG89] [Gel85] é o modelo mais representativo na realização do conceito de espaço partilhado, assentando num conjunto de operações simples (figura 3.6).

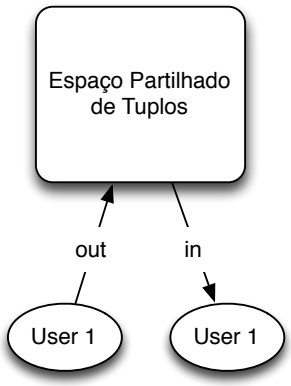

Figura 3.6: Espaço partilhado

Este modelo permite a coordenação e a comunicação entre diversos processos distribuídos que acedem a objectos que estão num espaço partilhado. A representação do espaço partilhado é alcançada através de um conjunto de tuplos. Um tuplo, neste contexto, é uma lista ordenada de campos de um determinado tipo, que podem ou não estar instanciados. A título de exemplo, apresenta-se o tuplo (a,b,c), que é um tuplo com três campos, no qual o primeiro campo tem o valor *a*, o segundo campo tem o valor *b* e o terceiro campo tem o valor *c*. Também é possível ter campos não instanciados nos tuplos, representados como "_".

De forma a aceder ao espaço de tuplos, o modelo define as seguintes operações principais, que podem ser invocadas concorrentemente:

- *out*, inserção de um tuplo no espaço partilhado. Esta operação é assíncrona e atómica. Como resultado desta operação, o tuplo é inserido no espaço partilhado e fica visível a todos os processos.

- *in* e *rd*, leitura atómica de um tuplo do espaço partilhado. A operação *in* efectua uma leitura destrutiva de um tuplo no espaço partilhado, enquanto que a operação *rd* devolve uma cópia de um tuplo, ou seja, efectua uma leitura não destrutiva. Estas duas operações são bloqueantes, isto é, o processo que efectuou a operação é bloqueado até que exista um tuplo no espaço partilhado que satisfaça o modelo do tuplo pedido. A verificação dessa condição é feita à custa de uma operação de *pattern matching*, em que o número de campos é comparado com os tuplos candidatos e cada um dos campos deve ser compatível com o valor submetido no tuplo [CG89].

A semântica das operações de actualização no modelo *Linda* são do tipo "*out-of-place*",

pois quando um processo quer actualizar o valor de um tuplo presente no espaço tem de primeiro remover esse tuplo através de uma operação *in*, para então alterar o seu valor e, posteriormente, inserir o tuplo de novo no espaço, através de uma operação *out*.

3.3.2 *JGroupSpace*

O desenvolvimento do modelo *JGroupSpace* [Cus08] teve como principal objectivo disponibilizar uma realização mais eficiente e flexível do modelo *GroupLog*, sem recorrer a uma linguagem de programação em lógica. Adicionalmente, visou distribuir o espaço partilhado de tuplos ao invés de centralizar na camada lógica (*Prolog*).

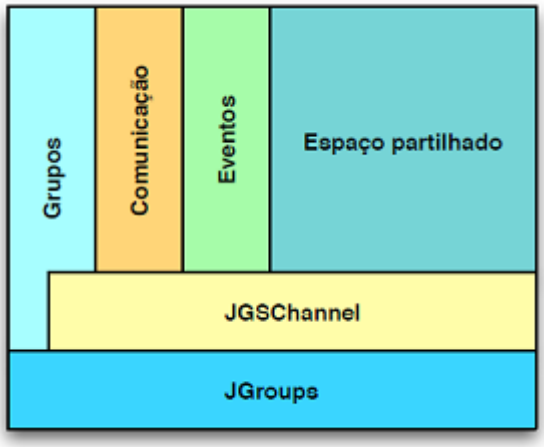

Figura 3.7: Arquitectura do modelo *JGroupSpace* [Cus08]

A implementação deste modelo foi totalmente concebida em *Java* e foi desenvolvida ao nível do *middleware*, de forma a permitir a sua utilização independentemente do sistema operativo de cada participante do sistema. A implementação assenta sobre o sistema de comunicação *JGroups* [Ban11] e adiciona um conceito que não existe na plataforma *JGroups*, o conceito de hierarquia de grupos. Quanto ao espaço partilhado de tuplos, este é suportado usando um conjunto de servidores dedicados que ficam associados a todos os grupos existentes no sistema. A semântica de acesso ao espaço partilhado segue as primitivas definidas no *GroupLog* bem como a adição de três primitivas presentes em extensões do modelo *Linda*, a operação *inp* e *rdp* [Car87], que têm o mesmo propósito que as operações *in* e *rd* respectivamente mas com uma semântica não bloqueante, e a operação *rdpall* que lê todos os tuplos presentes no espaço partilhado de tuplos.

3.3.3 *MAGO*

O modelo *MAGO* [Mor07] é um modelo inspirado no modelo *GroupLog*, que retém alguns dos aspectos essenciais desse modelo, em particular a definição de grupos e a existência de um espaço partilhado de tuplos por cada grupo, como uma forma de interacção entre os seus participantes. O modelo *MAGO* propõe um conjunto de primitivas, por forma a

suportar mecanismos de comunicação entre participantes, mais próximas das necessidades de apoio ao desenvolvimento de aplicações interactivas baseadas em grupos.

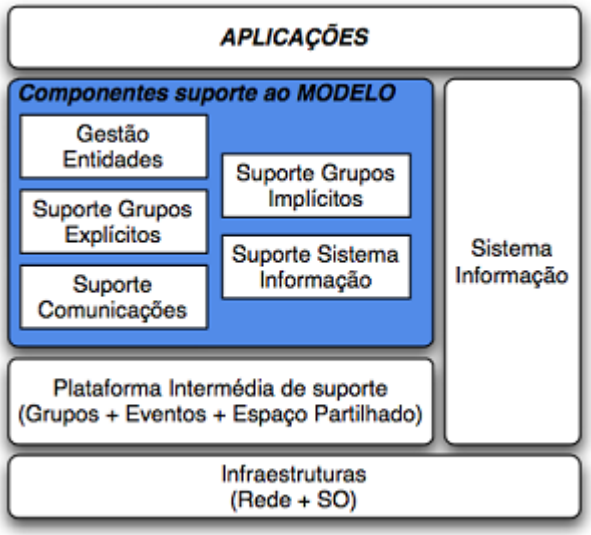

Figura 3.8: Arquitectura do modelo *MAGO* [Mor07]

Este modelo suporta comunicação directa entre os participantes, quer através de mensagens bem como invocação de métodos, e suporta comunicação de um para muitos, utilizando o mecanismo de eventos para difusão das mensagens enviadas para diversos participantes.

O modelo *MAGO* introduz o conceito de grupos implícitos, conceito não presente no modelo *GroupLog*, no qual é possível a criação automática de grupos de acordo com as preferências de cada participante. Por exemplo, se houver diversos participantes que têm nas suas preferências que gostam de viajar, então será formado um grupo para participantes que gostam de viajar.

Essas preferências são definidas ao nível da aplicação e são representadas como um conjunto de atributos, em que cada atributo representa um interesse do participante. Por fim, o modelo de espaço partilhado é utilizado em cada grupo com o intuito de partilha de informação entre os participantes desse mesmo grupo.

De forma a garantir persistência quanto a informação relevante ao sistema, como por exemplo as preferências mencionadas, o modelo permite a sua articulação com um sistema de informação persistente. Dependendo dos cenários e das necessidades da aplicação, a utilização deste sistema de informação é da responsabilidade da aplicação, sendo disponibilizada uma interface de programação que permite o acesso a um sistema de informação externo (ver figura 3.8).

3.4 Conclusão

Este capítulo discutiu alguns aspectos fundamentais inerentes aos modelos de grupos. Esses aspectos dividem-se nos seguintes: gestão de filiação e gestão de comunicação. De seguida, apresentaram-se dois exemplos de plataformas de apoio ao desenvolvimento de aplicações distribuídas, sendo ilustradas, para cada um deles, a sua arquitectura e o conjunto de primitivas básicas. No final do capítulo, discutiu-se um conjunto de modelos que integram o modelo de espaço partilhado no contexto de grupos: *GroupLog*, com uma implementação centralizada que recorre a uma linguagem de programação em lógica, sendo a concretização do espaço partilhado alcançada através do modelo *Linda*; *JGroupSpace*, implementação totalmente distribuída utilizando a linguagem de programação *Java* e *MAGO* que oferece um conjunto de primitivas mais vocacionadas para o desenvolvimento de aplicações interactivas.

Tendo em conta todos os modelos apresentados neste capítulo, desenvolveu-se a plataforma *Imagine* que incorpora alguns desses modelos e aspectos fundamentais de forma a disponibilizar um conjunto de primitivas que permite a implementação de jogos colaborativos.

4

Concepção da Plataforma *Imagine*

Após apresentadas as funcionalidades requeridas para o desenvolvimento de jogos colaborativos e apresentados modelos de grupos que respondem a algumas dessas funcionalidades, surge a motivação para a plataforma *Imagine*.

O nome da plataforma foi inspirado na música com o mesmo nome de *John Lennon*.

Neste capítulo descreve-se a plataforma *Imagine*, sendo inicialmente apresentados os objectivos e contribuições deste trabalho e de seguida uma descrição da plataforma, com a apresentação dos conceitos fundamentais e descrição das funcionalidades suportadas através das primitivas da interface de programação. Por fim, apresentam-se alguns exemplos que demonstram a utilização da plataforma.

4.1 Objectivos e Contribuições

A plataforma *Imagine* tem como principal objectivo facilitar a concepção e implementação de jogos colaborativos recorrendo a um modelo de grupos e à utilização do conceito de espaço partilhado, com o objectivo de facilitar a partilha de dados e a interacção entre os jogadores. A tabela 4.1 mapeia as três principais classes de funcionalidades requeridas pelos jogos colaborativos no referido modelo de grupos.

	Grupos	Comunicação por Mensagens	Espaço Partilhado
Organização	X		
Interacção	X	X	X
Dinamismo	X	X	

Tabela 4.1: Mapeamento de funcionalidades de jogos colaborativos sobre as funcionalidades da plataforma *Imagine*

A plataforma *Imagine* tem como propósito fornecer um nível de abstracção que permite aos programadores desenvolverem jogos colaborativos que ilustrem cenários de colaboração de forma simples, em que os detalhes de comunicação, gestão de participantes e grupos são tratados pela plataforma.

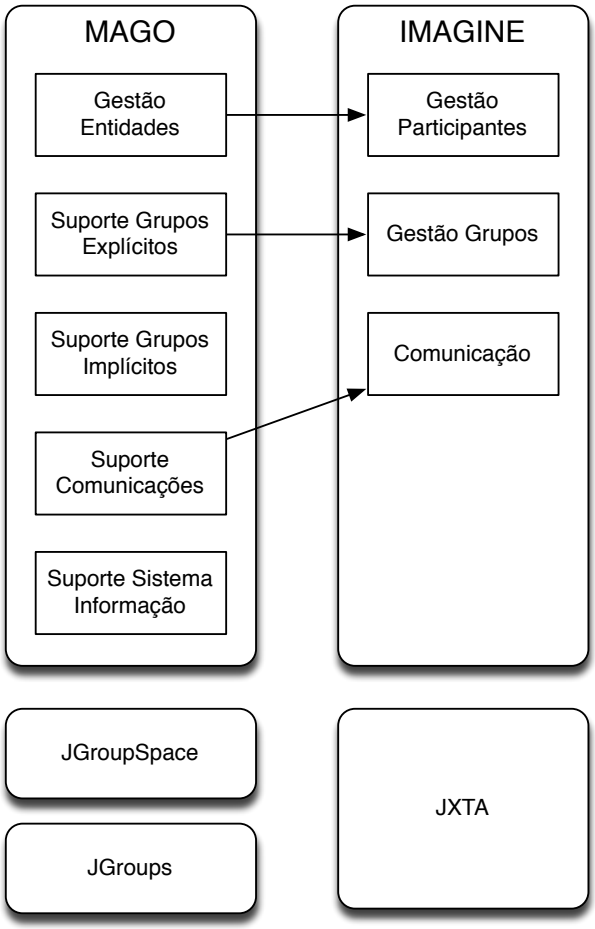

Figura 4.1: Ligação entre o modelo *MAGO* e a plataforma *Imagine*

A modelação dos grupos na plataforma *Imagine* é baseada no modelo *MAGO* (figura 4.1). Como base de implementação da plataforma *Imagine*, foi necessário escolher uma plataforma subjacente que tornasse disponível uma camada intermédia que suportasse o maior número de requisitos para a implementação do modelo, tendo sido escolhida para esse efeito a plataforma *JXTA*. Relativamente às funcionalidades disponibilizadas pelo modelo *MAGO*, como mencionado anteriormente, não se considerou necessário, neste trabalho, implementar todas as funcionalidades presentes nesse modelo, dado que algumas das funcionalidades presentes no modelo *MAGO* foram concebidas tendo em conta o desenvolvimento de aplicações que manipulam conteúdos multimédia.

A plataforma *Imagine* disponibiliza uma API em *Java*, que facilita o desenvolvimento de jogos colaborativos no que diz respeito ao suporte de primitivas de gestão e comunicação baseadas em grupos, estendido com o conceito de espaço de tuplos partilhados.

A colaboração nos diversos cenários presentes nesses jogos pode ser atingida através da utilização de um modelo de grupos por forma a facilitar a cooperação e comunicação entre diversos participantes.

De seguida apresenta-se a arquitectura da plataforma *Imagine* e descrevem-se os seus componentes.

4.2 Descrição da Plataforma *Imagine*

Os componentes do modelo *MAGO*, no qual a plataforma *Imagine* se baseia, são os seguintes: gestão de entidades; suporte de grupos explícitos e suporte a comunicações.

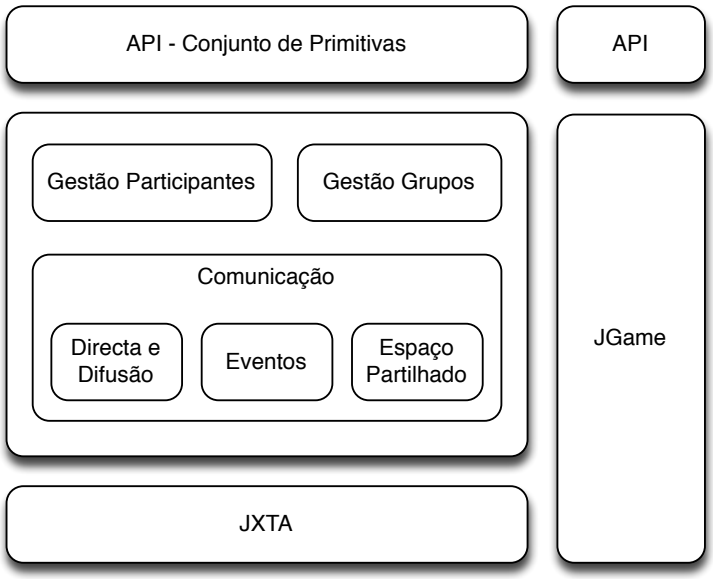

Figura 4.2: Arquitectura da plataforma *Imagine*

As primitivas disponíveis na API da plataforma *Imagine* suportam as seguintes classes de funcionalidade (figura 4.2):

- **Gestão de participantes**, responsável pela gestão de todos os participantes e respectiva informação associada a cada participante;
- **Gestão de grupos**, suportando todas as operações permitidas sobre grupos, envolvendo a criação e eliminação de grupos, bem como a gestão da sua filiação;
- **Comunicação**, dividida em três componentes:
  - **Comunicação directa e difusão**, referente à troca de mensagens directa entre participantes e à difusão de mensagens entre membros de um mesmo grupo;
  - **Eventos**, cuja concretização se baseia num modelo de publicação/subscrição;
  - **Espaço partilhado**, suportando as funcionalidades que permitem a manipulação de um espaço partilhado de tuplos, interno a cada grupo.

Relativamente ao desenho de interfaces para as aplicações desenvolvidas, é disponibilizado o acesso a um motor gráfico 2D, externo à plataforma (JGame, na figura 4.2), que possibilita o desenho de interfaces gráficas completas.

4.3 Conceitos Fundamentais

Esta secção apresenta as classes disponibilizadas na API, bem como os conceitos fundamentais presentes na plataforma e que são concretizados nas primitivas que a plataforma disponibiliza.

Como referido anteriormente, a plataforma *Imagine* baseia-se no modelo *MAGO* mas existem alguns conceitos que são adaptados e reutilizados de *JXTA* (por exemplo, a utilização de objectos de escuta *listeners* e a utilização de credenciais para a filiação de um participante num grupo), dado que esta serve como base de implementação da plataforma *Imagine* (figura 4.3).

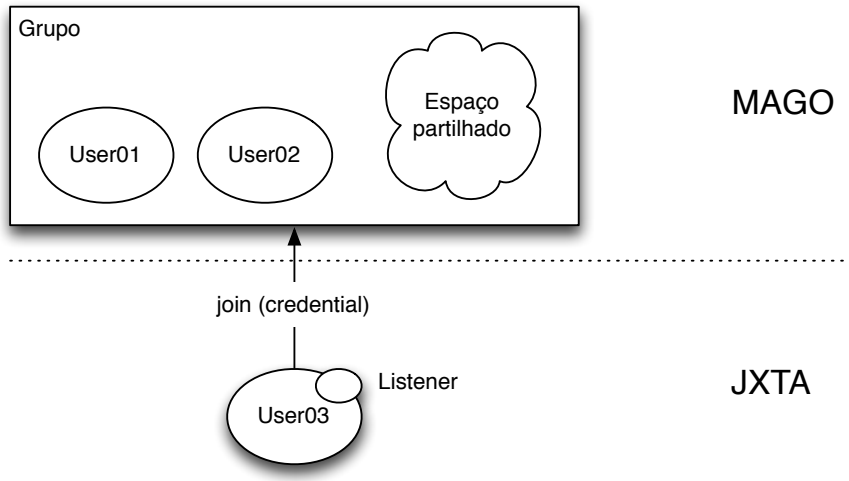

Figura 4.3: Aspectos incorporados do Modelo *MAGO* e de *JXTA* que se encontram presentes na plataforma *Imagine*

4.3.1 Organização da plataforma

Foram definidas as classes *Platform*, *Users*, *GroupsManager*, *Groups* e *JXTAGroup* que constituem a plataforma e permitem o acesso a todas a primitivas disponibilizadas pela plataforma (figura 4.4).

A classe *Platform* é o ponto de acesso, a partir da qual todas as primitivas são acedidas (figura 4.4). As primitivas dividem-se em dois grandes grupos, as primitivas de gestão de participantes e as de gestão de grupos, que são acedidas respectivamente através do objecto *users* (instanciação da classe *Users*) e do objecto *groups* (instanciação da classe *GroupsManager*).

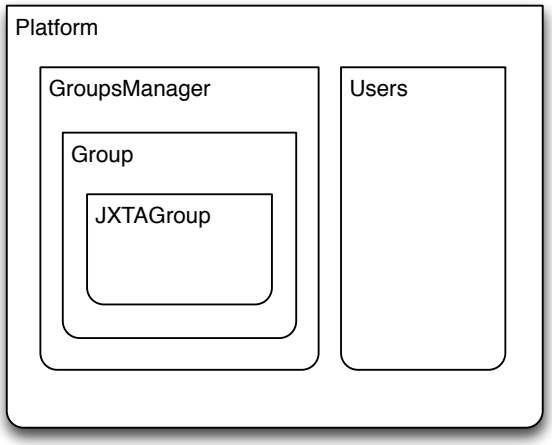

Figura 4.4: Classes presentes na plataforma *Imagine* e ligações entre elas

A gestão de participantes é alcançada através da classe *Users*, que contém os métodos de registo de um participante (`Users.register`) e eliminação do registo de um participante (`Users.unregister`), bem como as primitivas referentes ao envio de mensagens directas entre participantes (`Users.end`).

A gestão de grupos é concretizada através da classe *GroupsManager*. Esta classe incorpora as primitivas de gestão de grupos (`Groups.create`, `Groups.destroy`, `Groups.join`, `Groups.leave`), gestão de eventos por grupo (`Groups.advertise`, `Groups.publish`, `Groups.subscribe`, entre outras), acesso ao espaço partilhado por grupo (`Groups.sharedspace`) e comunicação entre participantes de um grupo (`Groups.send`). A classe *GroupsManager* tem um *Hash Map* que guarda todos os grupos nos quais o participante está filiado.

A representação de cada grupo é alcançada através da classe *Group*. Essa classe contém um *Hash Map* com todos os eventos que um participante subscreveu no contexto de cada grupo, bem como contém a classe *JXTAGroup* que representa o próprio grupo.

Essa última classe (*JXTAGroup*) contém a implementação, sobre *JXTA*, de todas as primitivas possíveis de efectuar num grupo.

4.3.2 Participantes

Um participante é representado na plataforma como membro de um grupo global, que abrange todos os participantes registados, dado que sempre que um participante se inicia na plataforma, fica associado a um grupo global. Este grupo global é uma característica da plataforma *JXTA* que oferece um ponto comum para todos os participantes.

A plataforma suporta esta entidade através da classe *Users*, sendo disponibilizadas as primitivas `Users.register` e `Users.unregister` para o registo e cancelamento do registo de um participante.

Relativamente à nomeação de um participante como entidade gerida ao nível da plataforma *Imagine*, é necessário registar explicitamente o nome que se pretende utilizar. Uma vez efectuado o registo com sucesso, o nome é único na plataforma e qualquer referência a esse nome identifica, de forma única, o respectivo participante.

Um participante existe desde o seu registo na plataforma até ao cancelamento desse registo. Nesse período de tempo pode efectuar quaisquer operações sobre a plataforma.

4.3.3 Grupos

Na plataforma, um grupo é identificado por um nome de tal modo, que à semelhança do mecanismo de nomeação existente para os participantes, os nomes associados aos grupos são sempre únicos.

Um grupo consiste numa organização lógica entre membros, podendo existir zero ou mais membros. As entradas e saídas num grupo ocorrem através da utilização de primitivas disponíveis para esse efeito. Todos os participantes registados na plataforma pertencem a um grupo global e todos os grupos criados na plataforma derivam deste grupo global, isto é, são subconjuntos.

A classe que gere todos os grupos é *GroupsManager*, sendo cada grupo suportado pela classe *Group*, que contém todos os eventos associados a este grupo e a representação do próprio grupo é alcançada através da classe *JXTAGroup*, que contém a implementação, sobre *JXTA*, das operações possíveis de efectuar num grupo.

Um grupo existe desde a sua criação (`Groups.create`) até à sua destruição (`Groups.destroy`). Enquanto o grupo existir é possível a qualquer participante filiar-se ao grupo (`Groups.join`) e é possível aos membros saírem do grupo (`Groups.leave`). Dentro do contexto de grupo também é possível enviar mensagens (`Groups.send`), bem como gerir os eventos (`Groups.publish`, `Groups.subscribe`, entre outros).

4.3.4 Comunicação

Um dos modelos de comunicação suportados pela plataforma é baseado em mensagens. O envio de mensagens é alcançado através da utilização de uma primitiva da API para esse efeito (`Groups.send` ou `Users.send`). Quanto à recepção de mensagens, essa é efectuada através de objectos de escuta (*listeners*), tendo por base o mecanismo existente em *JXTA* (secção 3.2.2).

No envio de uma mensagem para um participante ou grupo, é necessário indicar qual é o grupo, o nome e parâmetro desse objecto de escuta. Relativamente à recepção dessa mensagem, é necessário ter um objecto de escuta registado no mesmo grupo, com o nome e o parâmetro para o qual se enviou a mensagem.

Este mecanismo de troca de mensagens é um mecanismo reactivo dado que é criada uma instância de um objecto de escuta para todas as mensagens recebidas. Todas as mensagens tratadas por objectos de escuta são perdidas após a conclusão do tratamento

das mesmas, não existindo uma estrutura que guarde todas as mensagens recebidas ou tratadas até a um dado momento.

Quanto ao ciclo de vida de um objecto de escuta, este existe desde o seu registo, através da primitiva `Groups.addIncomingListener`, até que este seja removido explicitamente, através da primitiva `Groups.removeIncomingListener`, ou aquando da saída do participante do grupo no qual o objecto de escuta se encontra registado.

4.3.5 Eventos

A plataforma *Imagine* implementa um modelo de comunicação por eventos baseado num mecanismo de publicação/subscrição. Conforme apresentado na secção 3.1.3, este modelo possibilita criar e eliminar eventos. A criação ou eliminação de um evento consiste em manipular os tipos de eventos existentes, por exemplo, se um participante quiser publicar uma mensagem associada ao tema futebol, então deve publicar uma mensagem associada ao evento do tipo futebol.

A publicação de mensagens para um evento segue a metodologia presente no envio de mensagens, ou seja, o envio de mensagens é efectuado através de uma primitiva (`Groups.publish`). A recepção de mensagens é alcançada através de objectos de escuta, sendo que o objecto é registado no contexto do grupo que representa o evento, com um dado nome de objecto de escuta, que corresponde ao nome do evento criado e um dado parâmetro de objecto de escuta. A utilização de um parâmetro para os objectos de escuta associados a um evento, permite registar diversos objectos associados ao mesmo evento.

Um evento existe desde a sua criação, através da primitiva `Groups.advertise`, até à sua eliminação, através da primitiva `Groups.unadvertise`. Enquanto o evento existir, é possível divulgar mensagens sobre esse evento (`Groups.publish`, bem como subscrever ao evento (`Groups.subscribe`).

4.3.6 Espaço partilhado

O espaço partilhado implementado na plataforma *Imagine* segue o modelo presente no modelo *MAGO*, associando um espaço partilhado de tuplos a cada grupo. Para aceder ao espaço partilhado existe uma primitiva (`Groups.sharedspace`), na qual é possível efectuar as operações de manipulação do espaço partilhado (leitura (*find*,*consult* e *get*) e escrita (*update*)), dependendo do comando submetido como argumento (explicado em detalhe na secção 4.4.4).

Quanto à representação do espaço partilhado na plataforma, esta é alcançada através de um conjunto de tuplos, sendo cada tuplo definido como uma lista de campos ordenados. A título de exemplo, apresenta-se o tuplo (<a,b,c>), que representa uma lista de três campos, em que o primeiro campo contém o valor *a*, o segundo campo contém o valor *b* e o terceiro campo contém o valor *c*.

4.4 Funcionalidades Suportadas e Primitivas da Interface de Programação

Após apresentados os conceitos fundamentais presentes na plataforma, descrevem-se as primitivas que a mesma disponibiliza.

4.4.1 Gestão de participantes

A componente de gestão de participantes é responsável pelo registo e remoção de participantes na plataforma, sendo disponibilizadas duas primitivas para esse efeito.

4.4.1.1 Registo de um participante na plataforma

De forma a que um participante usufrua da plataforma, é necessário que este se registe na plataforma e obtenha um identificador único interno que permite aos restantes participantes tomarem conhecimento deste novo participante.

Listing 4.1: Registo de um participante na plataforma

```
1 boolean Users.register(name)
```

O registo de um participante na plataforma é efectuado através da primitiva `Users.register` fornecendo como argumento o nome pretendido. Esta primitiva retorna um valor de tipo booleano que corresponde ao resultado do registo do nome na plataforma. Caso o nome pretendido já se encontre em uso então o resultado da função é *false*, enquanto que, no caso contrário o registo deste participante é efectuado com sucesso sendo retornado *true*.

4.4.1.2 Remoção de um participante na plataforma

É dada a possibilidade a um participante de sair da plataforma a qualquer momento.

Listing 4.2: Remoção de um participante na plataforma

```
1 void Users.unregister()
```

Para remover um participante da plataforma deve ser utilizada a primitiva `Users.unregister`. Esta primitiva liberta o nome que estava associado ao participante que invocou a primitiva.

4.4.2 Operações sobre grupos

Face à componente de gestão de grupos, esta engloba todas as operações comuns de criação e eliminação de grupos, entrada e saída em grupos, bem como obtenção da lista dos membros de um grupo.

4.4.2.1 Criação de um grupo

A criação de um grupo pode ser efectuada por qualquer participante da plataforma, bastando fornecer o nome pretendido para o grupo.

Listing 4.3: Criação de um grupo

```
1 boolean Groups.create(name)
```

Relativamente à operação de criação de um grupo, esta é efectuada através da primitiva `Groups.create` sendo necessário fornecer como argumento o nome pretendido para o grupo.

Antes da criação do grupo com o nome pretendido, a plataforma verifica se o nome já se encontra reservado a outro grupo e se for este o caso, então a criação do grupo é abortada e é retornado *false*. Caso contrário é efectuada a criação do grupo sendo retornado *true*. Um grupo criado desta forma está inicialmente vazio, sendo necessário efectuar a chamada à primitiva `Groups.join` para um dado participante se tornar membro de um grupo (secção filiação num grupo 4.4.2.3).

Visto a plataforma *Imagine* assentar sobre *JXTA*, para definir uma política de acesso ao grupo é necessário estender o método de criação de um grupo, de forma a modificar o serviço de filiação (*MembershipService*) e adicionar a política de acesso pretendida.

4.4.2.2 Eliminação de um grupo

A eliminação de um grupo pode ser desencadeada por qualquer membro desse grupo e remove esse grupo da plataforma.

Listing 4.4: Eliminação de um grupo

```
1 void Groups.destroy(name, now)
```

A eliminação de um grupo é efectuada através da primitiva `Groups.destroy`, sendo necessário fornecer como argumento o nome do grupo e uma indicação de se a eliminação do grupo é imediata ou não.

No caso de uma eliminação imediata serão enviadas mensagens a todos os membros desse grupo avisando da sua eliminação. Cada membro que receba essa mensagem desencadeia a saída do grupo. No caso de uma eliminação não imediata, o grupo é marcado para eliminação até que todos os membros tenham saído do grupo.

Sempre que um participante efectua uma operação relativa a um grupo ou aos membros desse grupo, verifica se este está marcado para eliminação. Caso essa situação se verifique, então esse participante deixa de ser membro do grupo que está marcado para destruição.

A possibilidade de efectuar uma destruição imediata ou não de um grupo prende-se com o facto de dar a possibilidade de concluir operações que estão em curso ou não.

Numa destruição imediata todos os membros são avisados da destruição quando esta é iniciada e portanto efectuam a saída do grupo quando se apercebem da destruição

do mesmo, sendo que todas as operações que se encontravam em curso podem não ser concluídas.

Numa destruição não imediata, dado que só se verifica que um grupo encontra marcado para destruição quando se efectua uma operação explícita sobre um grupo, é possível concluir operações que se encontravam já em curso.

Em ambos os casos, após iniciado o processo de destruição de um grupo, todas as operações que sejam efectuadas sobre esse grupo dão erro.

4.4.2.3 Filiação num grupo

Um participante pode filiar-se em grupos previamente criados, sendo a sua admissão dependente da política de acesso associada ao grupo.

Listing 4.5: Filiação num grupo

```
1 void Groups.join(name, credential)
```

Para que um participante se torne membro de um grupo deve-se utilizar a primitiva `Groups.join`, sendo fornecido como argumento o nome do grupo ao qual se pretende filiar e uma credencial para validação do participante, caso seja necessário.

Do lado da plataforma, a filiação de um participante a um grupo passa por verificar se existe algum pré-requisito de filiação e efectuar o pedido de filiação ao grupo.

Caso seja necessário efectuar algum tipo de validação para filiação a um grupo, então é necessário o programador adicionar os dados necessários à credencial fornecida como argumento. Caso contrário, a plataforma cria uma credencial simples e submete essa credencial quando se efectua a filiação deste participante num determinado grupo. Caso o membro seja aceite no grupo é gerada uma credencial nova que identifica este participante como membro de um determinado grupo. Essa credencial é utilizada por *JXTA* para validar este participante como membro do grupo que se filiou.

Uma credential é definida como um documento *XML* que segue a estrutura base presente na listagem de código 4.6 e deve ser criada através do método `DiscoveryService.makeCredential` disponibilizado por *JXTA*.

Listing 4.6: Exemplo de uma credencial

```
1 <Cred>
2   <name>Player</name>
3   <signature>636da1d35e805b00eae0fcd8333f9234</signature>
4 </Cred>
```

4.4.2.4 Saída de um grupo

Um membro de um grupo pode a qualquer momento efectuar a saída desse grupo.

Listing 4.7: Saída de um grupo

```
1 void Groups.leave(name)
```

Para efectuar a saída de um membro de um grupo utiliza-se a primitiva `Groups.leave`, fornecendo como argumento o nome do grupo que se pretende abandonar.

4.4.2.5 Constituição de um grupo

Por fim, relativamente a operações sobre grupos, é possível obter a constituição de um grupo, pedindo uma lista com todos os membros correntes de um determinado grupo.

Listing 4.8: Constituição de um grupo

```
1 List<String> Groups.membership(name)
```

Essa lista é obtida através da primitiva `Groups.membership`, que desencadeia um processo de pesquisa de todos os membros associados ao grupo, sendo assim obtida uma lista contendo os nomes de todos os membros que estão actualmente activos no grupo e que responderam à pesquisa efectuada. Caso um membro não responda a esta pesquisa, por exemplo se esse membro não estiver disponível por problemas de rede, então esse membro não aparecerá na lista retornada.

4.4.2.6 Exemplo de operações sobre um grupo

Este exemplo (listagem de código 4.9) descreve a sequência de passos necessários à criação de um grupo, filiação no grupo criado e envio de uma mensagem para esse grupo, por parte de um participante.

Listing 4.9: Operações Sobre um Grupo

```
1 Users.register("user01");
2 success = Groups.create("group");
3 if(success)
4   Groups.join("group");
```

A primeira acção a efectuar é o registo do participante na plataforma (linha 1), fornecendo um nome que identifique este participante. Neste caso foi utilizado o nome "user01". De seguida e assumindo que o registo foi efectuado com sucesso de forma a simplificar este exemplo, cria-se um grupo com o nome "group" (linha 2). Caso o grupo tenha sido criado com sucesso e omo mencionado, um grupo quando é criado tem zero membros, então é necessário efectuar a entrada ao mesmo (linha 4).

4.4.3 Comunicação entre participantes

Os mecanismos de comunicação oferecidos pela plataforma *Imagine* dividem-se em três tipos: mensagens; eventos; espaço partilhado. Nesta secção é apresentada a componente relativa à comunicação directa entre participantes (*1-1*) e à difusão de mensagens por membros de um grupo (*1-n*).

Como mencionado anteriormente, existem alguns aspectos que são incorporados da plataforma *JXTA*. A estruturação das mensagens é um desses aspectos no qual o conteúdo

das mensagens é definido com uma lista de pares, em que cada par contém um conteúdo e uma chave que o identifica. Esta organização permite estruturar o conteúdo das mensagens e facilita o tratamento da mensagem, dado ser possível aceder directamente aos pares que se pretende manipular através das chaves desses pares.

A título de exemplo, apresenta-se a estrutura de uma mensagem que contém dois pares:

- <"From"; "User01»
- <"Welcome"; "Welcome new user!»

O conteúdo da mensagem apresentada consiste em dois pares, no qual o primeiro par tem como chave **From** e tem o conteúdo "User01" e o segundo par tem como chave **Welcome** e tem o conteúdo "*Welcome new user!*".

4.4.3.1 Comunicação directa

A comunicação directa entre dois participantes é alcançada através de uma de duas primitivas.

A primeira primitiva permite o envio de uma lista de conteúdos, que segue a estrutura apresentada anteriormente. A segunda primitiva é uma simplificação da primeira que permite o envio de um único par, evitando a criação explícita da lista de conteúdos.

Listing 4.10: Envio de diversos conteúdos numa mensagem directa

```
1 void Users.send(name, list, listenerName, listenerParam)
```

Face ao envio de uma mensagem directa entre dois participantes, esta é alcançada através da primitiva `Users.send` e tem os seguintes argumentos:

- *name*, nome do participante destinatário;
- *list*, lista de conteúdos a enviar;
- *listenerName*, qual o nome do objecto de escuta;
- *listenerParam*, qual o parâmetro do objecto de escuta.

Listing 4.11: Envio de um par numa mensagem directa

```
1 void Users.send(name, key, message, listenerName, listenerParam)
```

Esta segunda primitiva (`Users.send`) permite o envio de um único par numa mensagem, sendo que os argumentos *name*, *listenerName* e *listenerParam* têm o mesmo significado que a primitiva apresentada anteriormente e é necessário indicar qual é a chave e conteúdo a enviar através dos respectivos argumentos (*key* e *message*).

4.4.3.2 Difusão de mensagens por membros de um grupo

A difusão de mensagens pelos membros de um grupo segue a mesma abordagem inerente ao envio de mensagens directo entre participantes, isto é, existem duas primitivas para a difusão de mensagens por membros de um grupo. A primeira permite o envio de uma lista de conteúdos e uma segunda que permite o envio de um único par.

Apenas um membro do grupo pode enviar mensagens para esse grupo. Todos os membros irão receber essa mensagem mas apenas os membros que têm o objecto de escuta registado de forma correcta, isto é, associado ao grupo com o nome e parâmetro correcto, irão interpretar a mensagem.

Listing 4.12: Difusão de uma mensagem com diversos conteúdos para um grupo

```
1 void Groups.send(groupName, list, listenerName, listenerParam)
```

Para a difusão de mensagens pelos membros de um grupo está disponível a primitiva `Groups.send` e que conta com os seguintes argumentos:

- *groupName*, nome do grupo destino;
- *list*, lista de conteúdos a enviar;
- *listenerName*, qual o nome do objecto de escuta;
- *listenerParam*, qual o parâmetro do objecto de escuta.

Listing 4.13: Difusão de uma mensagem com um conteúdo para um grupo

```
1 void Groups.send(groupName, key, message, listenerName, listenerParam)
```

Esta segunda primitiva (`Groups.send`) simplifica o envio de um par numa mensagem, sendo que os argumentos *groupName*, *listenerName* e *listenerParam* têm o mesmo significado da primitiva anterior e é necessário fornecer a chave e conteúdo a enviar através dos argumentos *key* e *message*.

4.4.3.3 Recepção de mensagens

Como mencionado, a recepção de mensagens na plataforma *Imagine* é efectuada à custa de objectos de escuta e existem primitivas para registo e eliminação desses objectos.

Registo de um objecto de escuta

É possível registar objectos de escuta por cada participante, no contexto de um grupo com um determinado nome e parâmetro, de forma a que este consiga receber mensagens enviadas para si, quer mensagens directas como difusão de mensagens para um grupo em que este participante é membro.

As mensagens directas entre participantes são enviadas no contexto do grupo global e para manipular os objectos de escuta no contexto do grupo global basta submeter *null* no nome do grupo.

Listing 4.14: Registo de um objecto de escuta

```
1 void Groups.addIncomingListener(groupName, listener, listenerName,
      listenerParam)
```

De forma a registar um objecto de escuta para a recepção de mensagens, deve-se utilizar a primitiva `Groups.addIncomingListener`. Esta primitiva tem os seguintes argumentos:

- *groupName*, nome do grupo;
- *listener*, instância do objecto de escuta;
- *listenerName*, qual o nome do objecto de escuta;
- *listenerParam*, qual o parâmetro do objecto de escuta.

Relativamente ao nome de grupo submetido no registo de um objecto de escuta, se este for *null* então o objecto de escuta fica associado ao grupo global.

De seguida apresenta-se um exemplo de um objecto de escuta. Todos os objectos de escuta têm de estender a classe *EndpointListener* disponibilizada por *JXTA*.

Listing 4.15: Exemplo de um objecto de escuta

```
1 class Listener extends EndpointListener {
2   public void processIncomingMessage(Message msg, EndpointAddress srcAddr,
        EndpointAddress dstAddr) {
3     // Do something...
4   }
5 }
```

Como se pode verificar na listagem de código 4.15, um objecto de escuta sempre que recebe uma mensagem, tem sempre três argumentos definidos: a mensagem recebida; o endereço de quem enviou; o enderenço para o qual foi enviada a mensagem. Estes três argumentos são instanciados por *JXTA* aquando da recepção de uma mensagem.

Eliminação de um objecto de escuta

É dada a possibilidade de eliminar um objecto de escuta previamente registado, sendo esta funcionalidade particularmente interessante quando se pretende actualizar o funcionamento de um objecto de escuta, sendo necessário eliminar o antigo objecto de escuta e registar o novo.

Listing 4.16: Eliminação de um objecto de escuta

```
1 void Groups.removeIncomingListener(groupName, listenerName, listenerParam)
```

Para a eliminação de uma instância de um objecto de escuta previamente registado deve-se utilizar a primitiva `Groups.removeIncomingListener`.

Esta primitiva tem os seguintes argumentos:

- *groupName*, nome do grupo;
- *listenerName*, qual o nome do objecto de escuta;
- *listenerParam*, qual o parâmetro do objecto de escuta.

4.4.3.4 Exemplo de comunicação directa entre dois participantes

Nesta secção apresenta-se um pequeno exemplo do envio de uma mensagem do participante *user01* para o participante *user02* (ver figura 4.5).

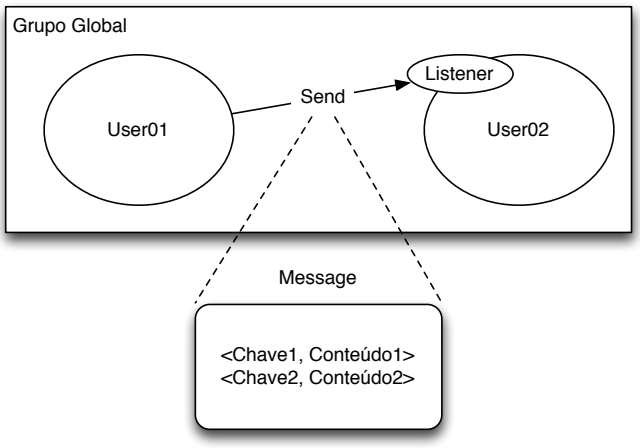

Figura 4.5: Envio de uma mensagem de um participante para outro, sendo a recepção efectuada através de um objecto de escuta

Listing 4.17: Exemplo que demonstra os passos essenciais ao envio e recepção de uma mensagem

```
1  // Contexto do user01
2  list = <"key1" -> "Content␣of␣the␣first␣key", "key2" -> "Content␣of␣the␣second␣
       key">
3  Users.send("user02", list, "listener", "");
4
5  // Contexto do user02
6  class Listener extends EndpointListener {
7    public void processIncomingMessage(Message msg, EndpointAddress srcAddr,
         EndpointAddress dstAddr) {
8      System.out.println(msg.getMessageElement("key1"));
9      System.out.println(msg.getMessageElement("key2"));
10   }
11 }
12 Groups.addIncomingListener(null, new Listener(), "listener", "");
```

O participante que envia a mensagem necessita apenas de invocar a primitiva `Users.send` (linha 3) e submeter os argumentos pretendidos para enviar a mensagem para o outro participante.

Quanto ao participante que irá receber a mensagem, este necessita registar um objecto de escuta (linha 12) que contém o tratamento da mensagem e neste exemplo, o objecto de escuta limita-se a imprimir o conteúdo da mensagem, sendo imprimido "Content of the first key" e "Content of the second key".

4.4.4 Gestão do espaço partilhado

Esta secção apresenta a componente relativa à gestão do espaço partilhado e as primitivas disponibilizadas para a sua manipulação.

Uma instância de um espaço partilhado está associada a um grupo e todas as operações efectuadas sobre esse espaço partilhado só podem ser efectuadas por membros desse grupo.

A organização de um espaço partilhado é alcançada através de um conjunto de tuplos. Um tuplo é definido como uma lista de campos ordenados. A pesquisa de tuplos no espaço partilhado é efectuada através de uma operação de *pattern matching*, que compara todos os tuplos presentes no espaço partilhado com o tuplo submetido na pesquisa.

Caso se pretenda aceitar num dado campo um qualquer valor, basta colocar no tuplo submetido esse campo com o valor "_", sendo que a pesquisa do lado do espaço partilhado interpreta esse valor como compatível com qualquer valor presente neste campo.

Todas as operações de manipulação de um espaço partilhado são convertidas no envio de mensagens directas para o gestor do espaço partilhado. Tendo em conta que o mecanismo de comunicação presente na plataforma é reactivo, é necessário indicar qual é o objecto de escuta que irá tratar da recepção da resposta da operação efectuada. Todos os objectos de escuta registados para o tratamento de respostas deste tipo têm de ser registados sobre o nome de *sharedspace*, no contexto do grupo global, ficando o parâmetro do objecto de escuta ao critério do programador.

É possível obter uma de duas respostas do gestor do espaço partilhado: uma resposta que contém apenas um tuplo de resposta e no qual os campos são identificados pelas chaves *field0*, *field1*, ..., *fieldN*; ou uma resposta que contém vários tuplos (M tuplos) e cada tuplo tem N_M campos, sendo os campos identificados pelas chaves, isto é:

- primeiro tuplo: *0field0*, ..., *0field*N_0;
- segundo tuplo: *1field0*, ..., *1field*N_1;
- tuplo M: *Mfield0*, ..., *Mfield*N_M.

Listing 4.18: Gestão do espaço partilhado

```
1 void Groups.sharedSpace(groupName, command, numOfFields, fields, wait, replyTo)
```

De forma a que um membro possa efectuar operações sobre o espaço partilhado de um grupo, é disponibilizada a primitiva `Groups.sharedSpace`. Existem quatro operações possíveis sobre o espaço partilhado, sendo a escolha da operação efectuada através do argumento *command*. A primitiva conta com os seguintes argumentos:

- *groupName*, nome do grupo;
- *command*, operação a executar no espaço partilhado, sendo possível executar um dos seguintes quatro comandos:
  - *find*, leitura não destrutiva de todos os tuplos que sejam válidos em comparação com o tuplo fornecido. Esta operação é não bloqueante e se o resultado desta operação for o conjunto vazio então a resposta contém esse conjunto.
  - *consult*, leitura não destrutiva de um tuplo que seja válido em comparação com o tuplo fornecido. Esta operação pode ser ou não bloqueante até que seja encontrado um tuplo que seja o pretendido, sendo esta opção indicada através do argumento booleano *wait*.
  - *get*, remoção (leitura destrutiva) de um tuplo do espaço partilhado que seja válido em comparação com o tuplo fornecido. Esta operação pode ser bloqueante até que exista o tuplo que se pretende remover, sendo esta opção indicada através do argumento booleano *wait*.
  - *update*, inserção de um tuplo no espaço partilhado. Esta operação é não bloqueante.
- *numOfFields*, número de campos presentes no tuplo;
- *fields*, lista de todos os campos associados ao tuplo;
- *wait*, indica se a operação é bloqueante ou não (caso seja possível aplicá-la sobre o comando indicado).
- *replyTo*, qual o parâmetro do objecto de escuta para o qual deve ser enviada a resposta. Se for submetido o valor *noreply* então o gestor do espaço partilhado não enviará resposta a este pedido.

4.4.4.1 Exemplo de gestão de um espaço partilhado

Nesta secção apresenta-se um exemplo (listagem de código 4.19) de gestão de um espaço partilhado que demonstra a inserção de um tuplo no espaço partilhado e posterior remoção e tratamento desse tuplo através de um objecto de escuta.

Listing 4.19: Operações Sobre um Espaço Partilhado

```
1 class Listener extends EndpointListener {
2   public void processIncomingMessage(Message msg, EndpointAddress srcAddr,
        EndpointAddress dstAddr) {
3     System.out.println(msg.getMessageElement("field0"));
4     System.out.println(msg.getMessageElement("field1"));
5   }
6 }
7
8 Object[] tuple = {"tuplefield", "date"}; // <"tuplefield", "date">
```

```
9  Groups.sharedSpace("group", "update", 2, tuple, false, "noreply");
10
11 Groups.addIncomingListener("group", new Listener(), "sharedspace", "printer");
12
13 Object[] tuple = {"_", "_"}; // <"_", "_">
14 Groups.sharedSpace("group", "get", 2, tuple, true, "printer");
```

Inicialmente é criado um tuplo com dois campos (linha 8), no qual o primeiro campo tem o valor "*tuplefield*" e o segundo tem o valor "*date*". Esse tuplo é inserido no espaço partilhado associado ao grupo *group* (linha 9), sendo indicado como argumento que o comando a efectuar é *update*, que pretende inserir um tuplo com os dois campos declarados anteriormente, indicação de que esta operação é não bloqueante e por fim indicação de que não se pretende obter resposta desta operação.

Após efectuada a inserção, é registado um objecto de escuta no mesmo grupo (*group*) (linha 11) sobre o nome de objecto de escuta *sharedspace* e parâmetro *printer*. Esse objecto de escuta (linha 1 a 6), imprime os dois primeiros campos de um tuplo. Dado que se pretende remover o tuplo inserido e a remoção só retorna um tuplo, então as chaves que identificam o tuplo são *field0* e *field1*.

Por fim, efectua-se um pedido de remoção de um tuplo com dois campos, em que esses dois campos podem conter qualquer valor (linha 13). Na invocação da primitiva (linha 14) indica-se que a remoção é bloqueante e que a resposta tem de ser encaminhada para o objecto de escuta que tem o parâmetro *printer*.

4.4.5 Gestão de eventos

A terceira e última componente associada à comunicação consiste na gestão de eventos.

Como mencionado anteriormente, o modelo de comunicação baseado em eventos aqui implementado assenta num mecanismo de publicação/subscrição que possibilita a qualquer membro de um grupo, criar eventos associado a um grupo e publicar mensagens associadas a esse evento. Como por exemplo, criar um evento com o nome "saídas" para reportar todas as saídas de membros de um determinado grupo, no qual cada membro que pretende sair, é responsável por publicar um mensagem a informar que este membro vai sair do grupo.

Não existem eventos pré-definidos na plataforma (por exemplo, publicação de uma mensagem sempre que há uma nova filiação num grupo), sendo a gestão deste tipo de eventos da responsabilidade do programador.

As operações que são possíveis de efectuar quanto à gestão de eventos seguem o mesmo esquema das operações sobre grupos, visto o modelo de publicação/subscrição ter correspondência, nas operações disponibilizadas por este, com o modelo de grupos apresentado.

Quanto à publicação de mensagens e tendo em conta o mecanismo de comunicação presente na plataforma *Imagine*, a publicação de mensagens é efectuada através da invocação da primitiva `Groups.publish`. Quanto à recepção dessas mensagens, essa é

efectuada através da utilização de objectos de escuta.

4.4.5.1 Criação de um evento

É dada a possibilidade a qualquer membro de um grupo de criar um evento associado ao grupo, ficando assim disponível um novo evento para a publicação de mensagens. Cada evento tem um nome único, no contexto de cada grupo.

Listing 4.20: Criação de um evento

```
1 boolean Groups.advertise(groupName, eventName)
```

A criação de um evento é efectuada através da primitiva `Groups.advertise`, na qual é necessário fornecer como argumento qual o nome do grupo (*groupName*) e qual o nome do evento (*eventName*). Caso o nome do evento pretendido já exista, então a primitiva retorna *false*; caso contrário, a primitiva retorna *true*.

4.4.5.2 Eliminação de um evento

É possível a qualquer membro de um grupo eliminar eventos associados ao grupo, indicando o nome do evento, como argumento da primitiva.

Listing 4.21: Eliminação de um evento

```
1 void Groups.unadvertise(groupName, eventName)
```

Caso se pretenda eliminar um evento, por exemplo, no caso de um determinado evento se tornar inválido e não forem publicadas mais mensagens associadas a esse evento, existe a primitiva `Groups.unadvertise`, na qual é necessário fornecer o nome do grupo (*groupName*) e o nome do evento (*eventName*).

4.4.5.3 Subscrição de um evento

Após criado um evento associado a um grupo, é possível a qualquer membro desse grupo subscrever esse evento, ficando assim a receber todas as mensagens publicadas associadas a esse evento, desde a sua subscrição.

Listing 4.22: Subscrição de um evento

```
1 void Groups.subscribe(groupName, eventName)
```

A subscrição a um evento é efectuada através da primitiva `Groups.subscribe`, na qual é necessário fornecer como argumento, qual o nome do grupo (*groupName*) e qual o nome do evento (*eventName*).

4.4.5.4 Anular a subscrição de um evento

Um subscritor de um dado evento pode a qualquer momento anular a sua subscrição e todos as mensagens associadas a esse evento que forem geradas a partir daí, não serão recebidas por este participante.

Listing 4.23: Anular a subscrição de um evento

```
1 void Groups.unsubscribe(groupName, eventName)
```

A anulação da subscrição de um evento é efectuada através da primitiva `Groups.unsubscribe`, sendo necessário fornecer como argumento, qual o nome do grupo (*groupName*) e o nome do evento (*eventName*) ao qual se pretende anular a subscrição.

4.4.5.5 Publicação de mensagens associadas a um evento

Qualquer membro de um grupo pode publicar mensagens associadas a um evento.

Listing 4.24: Publicação de uma mensagem com diversos conteúdos num evento

```
1 void Groups.publish(groupName, list, eventName, listenerParam)
```

Quanto à publicação de mensagens associadas a um evento, é disponibilizada a primitiva `Groups.publish`, que à semelhança da primitiva `Groups.send`, é dividida em duas primitivas, uma primeira que permite o envio de uma lista de conteúdos e uma simplificação dessa primitiva de forma a enviar um único par de conteúdo.

A primeira primitiva aceita os seguintes argumentos:

- *groupName*, nome de grupo a que o evento está associado;
- *list*, lista de conteúdos a enviar;
- *eventName*, nome do evento;
- *listenerParam*, parâmetro do objecto de escuta para o qual se pretende enviar a mensagem.

Listing 4.25: Publicação de uma mensagem com um conteúdo num evento

```
1 void Groups.publish(groupName, key, message, eventName, listenerParam)
```

Para a publicação de uma mensagem com apenas um par de conteúdo, existe uma simplificação da primitiva `Groups.publish`, no qual os argumentos *groupName*, *eventName* e *listenerParam* têm o mesmo significado que na primitiva anterior e é necessário indicar a chave e conteúdo que se pretende enviar.

4.4.5.6 Recepção de mensagens associadas a um evento

De forma a receber as mensagens associadas a um evento e tendo em conta o mecanismo de comunicação presente na plataforma *Imagine*, é necessário registar um objecto de escuta associado a esse evento.

Listing 4.26: Registo de um objecto de escuta associado a um evento

```
1 void Groups.addIncomingEventListener(groupName, listener, eventName,
      listenerParam)
```

Esse registo é efectuado através da primitiva `Groups.addIncomingEventListener`. A primitiva de registo de um objecto de escuta associado a um evento recebe os seguintes argumentos:

- *groupName*, nome do grupo a que o evento está associado, não sendo possível submeter este argumento como vazio, visto um evento estar sempre associado a um grupo;
- *listener*, instância do objecto de escuta;
- *eventName*, nome do evento, que corresponde ao nome do objecto de escuta;
- *listenerParam*, parâmetro do objecto de escuta;

Pode haver a necessidade de eliminar objectos de escuta associados a eventos, por exemplo, quando se pretende actualizar o comportamento de um determinado objecto de escuta. Para tal, é necessário remover o objecto de escuta antigo e registar o novo.

Listing 4.27: Eliminação de um objecto de escuta associado a um evento

```
1 void Groups.removeIncomingEventListener(groupName, eventName, listenerParam)
```

A eliminação de um objecto de escuta associado a um evento é efectuada através da primitiva `Groups.removeIncomingEventListener`.

Esta primitiva remove a instância do objecto de escuta associado aos argumentos submetidos. Segue-se um resumo dos argumentos:

- *groupName*, nome do grupo a que o evento está associado, não sendo possível submeter este argumento como vazio, visto um evento estar sempre associado a um grupo;
- *eventName*, nome do evento, que também corresponde ao nome do objecto de escuta;
- *listenerParam*, parâmetro do objecto de escuta.

4.4.6 Gestão de grupos e eventos de forma persistente

Na plataforma *Imagine* existe um mecanismo para salvaguardar informação sobre todos os grupos e respectivos eventos a que um participante se filiou explicitamente, sendo essa informação salvaguardada de forma persistente. Em concreto, são guardadas em ficheiro as informações relativas a todos os grupos e respectivos eventos de forma a que seja possível filiar-se mais tarde aos mesmos.

Não são guardados os objectos de escuta registados para os grupos e eventos, dado que estes têm implementação própria e seria necessário registar essa implementação de forma persistente.

Listing 4.28: Salvaguardar os grupos e eventos a que um participante está associado

```
1 void Platform.save()
```

De forma a recuperar o último estado salvaguardado que o participante guardou de forma persistente, disponibiliza-se a primitiva `Platform.load`. Essa primitiva lê o ficheiro que contém a lista de todos os grupos e respectivos eventos e tenta filiar-se a cada um deles, ou seja, a plataforma tenta filiar o participante em todos os grupos indicados no ficheiro e tenta subscrever todos os eventos associados ao grupos, que o participante tinha subscrito.

Listing 4.29: Recuperar os grupos e eventos a que um participante está associado

```
1 void Platform.load()
```

Por fim, existe uma primitiva auxiliar que actualiza o conteúdo da *cache* interna no contexto de um grupo (`Groups.updateCache`). Esta *cache* guarda informação relativa a grupos ou participantes que se encontram ao longo da execução da plataforma e é discutida no capítulo seguinte na secção 5.3.1.2.

Esta primitiva efectua uma pesquisa por todos os grupos no contexto de um grupo, por exemplo eventos criados associados a esse grupo. Esta primitiva é particularmente interessante dado que ao manipular-se um sistema distribuído, por vezes é necessário iniciar uma pesquisa activa de forma a descobrir todos os grupos associados a um contexto.

Listing 4.30: Actualizar a *cache* interna

```
1 void Groups.updateCache(groupName)
```

4.5 Conclusão

Nesta capítulo apresentou-se a plataforma *Imagine*, quais as funcionalidades suportadas e a sua interface de programação. Ao longo da descrição da interface de programação, foram ilustrados pequenos exemplos de utilização das primitivas.

A tabela 4.2 resume quais são as primitivas da plataforma *Imagine*.

Após apresentada a plataforma do ponto de vista do programador, discutem-se detalhes de implementação que foram tidos em consideração na implementação do protótipo da plataforma.

Gestão de participantes	
Registo de um participante	`Users.register`
Remoção de um participante	`Users.unregister`
Gestão de grupos	
Criação de um grupo	`Groups.create`
Eliminação de um grupo	`Groups.destroy`
Filiação num grupo	`Groups.join`
Saída de um grupo	`Groups.leave`
Constituição de um grupo	`Groups.membership`
Envio de mensagens	
Envio de uma mensagem directa	`Users.send`
Difusão de mensagens num grupo	`Groups.send`
Recepção de mensagens	
Registo de um o.e.	`Groups.addIncomingListener`
Eliminação de um o.e.	`Groups.removeIncomingListener`
Gestão do espaço partilhado	
Gestão do espaço partilhado	`Groups.sharedSpace`
Gestão de eventos	
Criação de um evento	`Groups.advertise`
Eliminação de um evento	`Groups.unadvertise`
Subscrição de um evento	`Groups.subscribe`
Anular a subscrição de um evento	`Groups.unsubscribe`
Publicação de mensagem num evento	`Groups.publish`
Registo objecto escuta num evento	`Groups.addIncomingEventListener`
Eliminação objecto escuta num evento	`Groups.removeIncomingEventListener`
Gestão de grupos e eventos de forma persistente	
Salvaguardar os grupos e eventos	`Platform.save`
Recuperar os grupos e eventos	`Platform.load`
Actualizar a *cache* interna	`Groups.updateCache`

Tabela 4.2: Resumo das primitivas disponibilizadas

5

Concretização da Plataforma *Imagine*

Este capítulo apresenta as opções de concepção tomadas na construção da plataforma *Imagine*. Após a discussão das opções de concepção, descreve-se como esta foi implementada, sendo analisadas em detalhe as primitivas disponibilizadas e qual o seu mapeamento na plataforma subjacente.

5.1 Discussão de Opções de Concepção

Como descrito no capítulo anterior, pretendia-se implementar uma plataforma[1] que disponibilizasse primitivas de gestão de participantes e grupos, bem como comunicação entre participantes.

A plataforma de suporte utilizada na implementação da plataforma *Imagine* foi *JXTA*, anteriormente descrita e analisada na secção 3.2.2. Em concreto, foi utilizada uma implementação de *JXTA* cuja API está disponível na linguagem de programação *Java*.

JXTA não oferece primitivas quanto à gestão de eventos, nem tem incorporado o conceito de espaço partilhado, tendo estes sido implementados ao nível da plataforma *Imagine*.

Quanto à componente gráfica presente ao nível da arquitectura da plataforma *Imagine*, esta é concretizada através da integração de um motor gráfico externo 2D. O motor escolhido foi o *JGame*. Dos diversos motores gráficos testados, este destaca-se pela sua facilidade, simplicidade de utilização para a elaboração de interfaces completas e integração fácil na plataforma, devido à sua implementação na linguagem de programação *Java*.

[1] Sempre que se menciona plataforma neste capítulo, refere-se à plataforma *Imagine*.

5.2 Descrição do Mapeamento da Arquitectura da Plataforma *Imagine* sobre *JXTA*

O mapeamento da arquitectura da plataforma *Imagine* sobre *JXTA* foi concretizado em duas fases. A primeira fase reflecte a utilização directa dos métodos oferecidos pelo sistema *JXTA* e que permitem a disponibilização das primitivas referentes à gestão de participantes, grupos e comunicação inerente a estes. A segunda fase foi composta pela implementação da gestão de eventos e gestão do espaço partilhado à custa das primitivas já implementadas na primeira fase.

A figura 5.1 resume o mapeamento dos conceitos presentes na plataforma *Imagine* sobre *JXTA*. Um participante é representado com a classe *User* na plataforma *Imagine* e é representado com a classe *PeerGroup* em *JXTA*. Os grupos são representados com a classe *Group* na plataforma e são representados com a classe *PeerGroup* em *JXTA*, sendo que os eventos são modelados na plataforma *Imagine* à custa de subgrupos. Os restantes conceitos presentes na plataforma (eventos e espaço partilhado) são implementados à custa das primitivas disponibilizadas pela plataforma *Imagine*.

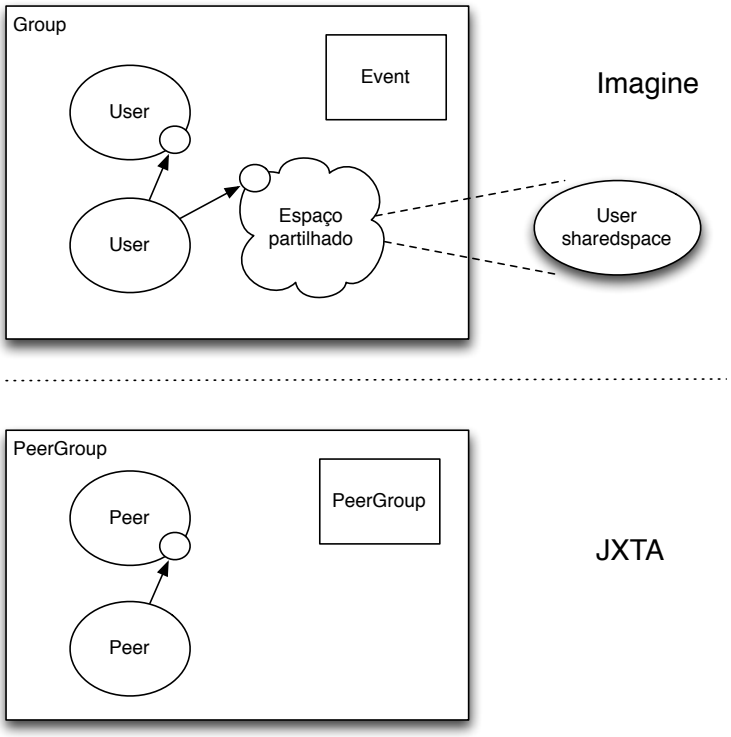

Figura 5.1: Mapeamento da arquitectura da plataforma *Imagine* sobre *JXTA*

De seguida, analisa-se cada componente da plataforma e resume-se o mapeamento efectuado para cada componente.

5.2.1 Participantes

Um participante é identificado por um nome único na plataforma e esse participante é membro de um grupo global, que abrange todos os participantes registados.

A plataforma suporta esta entidade através da classe *Users*. As operações disponibilizadas por esta classe que permitem o registo (`Users.register`) e o cancelamento do registo de um participante (`Users.unregister`), manipulam a classe base de *JXTA*, a classe *NetworkManager*.

A gestão de participantes é concretizada directamente utilizando a política que *JXTA* definiu para a gestão destes. Cada participante tem um identificador de sistema, sendo este gerado a partir do nome fornecido no registo do participante. Esse identificador de sistema é uma característica fundamental de *JXTA*, que permite que um participante possa alterar o endereço do dispositivo físico onde está a ser executado e mantenha o identificador de sistema, mantendo assim a transparência na localização.

A classe *Peer* presente em *JXTA* serve para referenciar outros participantes, sendo esta bastante útil para o envio de mensagens, por exemplo.

Um participante tem sempre um anúncio que o identifica. Isto simplifica a tarefa de reconhecimento de novos participantes, sendo apenas necessário distribuir o anúncio que o identifica. O conceito de anúncio é definido em *JXTA* e foi introduzido na secção 3.2.2.3.

JXTA permite o registo de participantes com o mesmo nome dado que *JXTA* identifica os participantes com identificadores únicos gerados a partir do nome submetido. Na plataforma *Imagine* o nome de cada participante é único e proibe-se o registo de participantes com o mesmo nome.

5.2.2 Grupos

Um grupo é identificado por um nome e esse nome é único na plataforma. Cada grupo consiste numa organização lógica entre membros, na qual podem existir zero ou mais membros e onde as entradas e saídas desse grupo ocorrem através da utilização de primitivas para esse efeito. Todos os participantes do sistema pertencem a um grupo global e todos os grupos criados na plataforma derivam deste grupo global, definindo assim uma hierarquia de grupos na qual a raiz é o grupo global.

A classe que representa um grupo na plataforma *Imagine* é a classe *Group*. Esta classe contém um *Hash Map* com todos os eventos que um participante subscreveu no contexto desse grupo e contém a classe *JXTAGroup* que representa o próprio grupo. Essa classe *JXTAGroup* contém a implementação, sobre *JXTA*, das operações possíveis de efectuar num grupo.

A classe *Group* oferece uma organização ao conceito de grupo e subgrupos (eventos) que existem no contexto deste grupo. Todas as primitivas disponibilizadas por esta classe têm ligação directa com as primitivas presentes na classe que representa o grupo (*JXTAGroup*) ou no subgrupo (evento) que se pretende manipular (que por sua vez também é

representado como um objecto *JXTAGroup*).

A classe *JXTAGroup* contém a implementação, sobre *JXTA*, das operações de gestão de grupos. Estas operações manipulam alguns dos serviços disponibilizados por *JXTA*, como por exemplo, o serviço de filiação (*MembershipService*), o serviço de comunicação (*EndpointService*) e o serviço de descoberta (*DiscoveryService*). Esses serviços são disponibilizados pela classe *PeerGroup*, que representa em *JXTA* um grupo.

Um grupo tem sempre um anúncio que o identifica e que à semelhança dos participantes, conta com as mesmas propriedades apresentadas na secção 3.2.2.3.

5.2.3 Comunicação

Um dos modelos de comunicação suportados pela plataforma é baseado em mensagens, sendo este ligado directamente com o mecanismo existente em *JXTA*. O envio de mensagens é alcançado através da utilização de uma primitiva da API para esse efeito (`Groups.send` ou `Users.send`). Quanto à recepção de mensagens, essa é efectuada através de objectos de escuta (*listeners*).

No envio de uma mensagem para um participante ou grupo, é necessário indicar qual é o grupo, o nome e parâmetro desse objecto de escuta. Relativamente à recepção dessa mensagem, é necessário ter um objecto de escuta registado no mesmo grupo, com o nome e o parâmetro para o qual se enviou a mensagem.

Para cada mensagem recebida por um participante, é efectuada uma pesquisa de um objecto de escuta que esteja registado para a recepção da mensagem. Se existir esse objecto de escuta, então é lançada uma instância desse objecto para tratar essa mensagem (idêntico ao conceito de *handler*). Esses objectos de escuta derivam da classe *EndpointListener* disponibilizada por *JXTA*.

5.2.4 Eventos

A gestão de eventos é alcançada através do modelo publicação/subscrição, apresentado na secção 3.1.3. Este modelo é adaptado no sistema *JXTA* através da criação de subgrupos associados ao grupo ao qual se pretende criar um evento, ou seja, quando se pretende criar um evento, este está sempre associado a um grupo e portanto será criado um subgrupo que representa esse tipo de evento. Todos os participantes que pretendem subscrever esse evento necessitam de entrar nesse subgrupo e registar um objecto de escuta para que possa receber todas as mensagens associadas a esse evento.

5.2.5 Espaço partilhado

Face ao espaço partilhado, este é implementado em dois contextos, o de cliente e o do servidor.

Do lado do cliente, para comunicar com o espaço partilhado existe a primitiva `Group.sharedSpace` que oferece as quatro operações descritas na secção 3.3. Essa primitiva encarrega-se de enviar o pedido para o gestor do espaço partilhado com os respectivos

argumentos passados aquando da invocação. Para a recepção das respostas dos pedidos efectuados ao espaço partilhado, é necessário registar um objecto de escuta com o nome *sharedspace* e parâmetro à escolha do programador, explicado em detalhe na secção 5.3.4.

Do lado do servidor e como já foi referido, a implementação do espaço partilhado visou comprovar a utilidade deste conceito, no âmbito da proposta desta dissertação, pelo que se adaptou uma concretização simplificada do conceito de espaço partilhado através da utilização de um único servidor que gere os espaços partilhados associados a todos os grupos. Este servidor é representado na plataforma como um participante com o nome *sharedspace*. Este servidor cria um novo espaço partilhado sempre que um grupo é criado, sendo para esse efeito gerada uma mensagem, por forma a avisar o servidor de que existe um novo grupo. Todas as operações que são invocadas sobre o espaço partilhado são convertidas em mensagens directas para o participante *sharedspace*.

Ao implementar-se o espaço partilhado desta forma, possibilitou-se um mais rápido desenvolvimento do mesmo de forma a testar e validar todos os seus aspectos. Em alternativa, seria possível implementar o espaço partilhado distribuindo os seus tuplos pelos participantes que estão envolvidos nesse grupo e utilizando um mecanismo de gestão da coerência de informação entre todos os participantes.

5.3 Implementação das Primitivas da Interface de Programação

Após apresentadas as linhas gerais que suportam a concretização da plataforma, descreve-se em pormenor a implementação de cada primitiva, sob a forma de pseudo-código, para ilustrar os diferentes passos presentes na sua execução.

As primitivas são apresentadas através da mesma organização utilizada na secção 4.4.

5.3.1 Gestão de participantes

A gestão de participantes é concretizada através da classe *Users* que disponibiliza dois métodos e que correspondem às seguintes primitivas: registo de um participante (`Users.register`) e cancelamento de um participante na plataforma (`Users.unregister`).

5.3.1.1 Registo de um participante na plataforma

A primitiva de registo de um participante na plataforma (listagem de código 5.1) verifica se o nome pretendido já está reservado (linha 2 e 3), utilizando a função privada `JXTAGroup.getAdvs` para procurar o anúncio de um participante com o nome pretendido. Esta função privada é invocada sobre o grupo global e permite a pesquisa de anúncios relativos a participantes (explicado de seguida)

Listing 5.1: Registo de um participante na plataforma

```
1 public boolean Users.register(name) {
2   List res = Platform.worldGroup.getAdvs(DiscoveryService.PEER, "Name", name);
3   if(!res.empty()) {
```

```
4       Platform.start(name);
5       return true;
6     } else {
7       return false;
8     }
9   }
```

Se não existe tal anúncio então inicia-se a plataforma, através da função privada *Platform.start*, com o nome pretendido e retorna-se *true* (linha 4); caso contrário retorna-se *false* como resultado desta função, sendo necessário invocar a função novamente, com um nome diferente no argumento, para efectuar o registo do participante no sistema.

Pesquisa de anúncios na plataforma

Tendo em conta o mecanismo de anúncios presentes em *JXTA* e dado que cada participante e grupo tem um anúncio que o identifica, é necessário disponibilizar uma função, ao nível da plataforma, que utilize o mecanismo de procura de anúncios no contexto de um grupo. A função `JXTAGroup.getAdvs` dá essa possibilidade, bastando indicar qual o tipo de anúncio que se procura (se é um anúncio associado a um participante, a um grupo ou um anúncio geral), qual o atributo que se procura e o valor desse atributo.

Listing 5.2: Procura de anúncios

```
1 private List<Advertisement> JXTAGroup.getAdvs(advType, attribute, value) {
2   discoveryService.getRemoteAdvertisements(advType, attribute, value);
3   return discoveryService.getLocalAdvertisements(advType, attribute, value);
4 }
```

A função privada `JXTAGroup.getAdvs` (listagem de código 5.2) efectua inicialmente uma pesquisa remota (linha 2), isto é, difunde-se a pesquisa por todos os participantes, na qual os argumentos submetidos são os seguintes: *advType* contém o tipo de anúncio pretendido, sendo possível procurar por participante, grupo ou anúncio genérico; *attribute* reflecte o tipo de atributo que se pretende efectuar na pesquisa, como por exemplo *Name*; *value* contém o valor que se pretende procurar.

Após efectuada a pesquisa remota, que é invocada de forma assíncrona, não bloqueando o fluxo de execução do método invocador, efectua-se uma pesquisa na *cache* (a *cache* é explicada na secção 5.3.1.2) com os mesmos parâmetros da pesquisa remota (linha 3), sendo o resultado obtido desta pesquisa local o resultado pretendido.

Visto a pesquisa de anúncios depender dos restantes participantes do sistema *JXTA* é necessário ter em atenção que é possível que após executada uma pesquisa remota, o resultado obtido não reflicta o estado actual global de *JXTA*, visto ser impraticável obter de forma precisa o estado global de um sistema distribuído.

Iniciar a plataforma

A classe *Platform* disponibiliza uma função privada que contém a sequência de acções necessária para iniciar *JXTA*.

Listing 5.3: Iniciar a plataforma

```
1 private void Platform.start(Mode, name) {
2   managerName = name.equalsIgnoreCase("")?"undefined":name;
3   JXTA.startNetwork(Mode, managerName);
4   worldGroup = JXTA.worldGroup;
5   Platform.load()
6 }
```

Como ilustrado na listagem de código 5.3 que representa a iniciação da plataforma, quando não se fornece nenhum nome como argumento define-se o nome deste participante como *undefined* (linha 2). Este nome reflecte um participante que ainda não está registado no sistema, ficando o nome de participante *undefined* reservado para ser utilizado pela plataforma para reflectir esse estado.

Para iniciar *JXTA*, é necessário referir qual o modo de execução. Esses modos foram descritos na secção 3.2.2. Na implementação utilizada do sistema *JXTA* existem os seguintes modos:

- *ADHOC*: normalmente associado a um participante que não tem recursos para oferecer funcionalidades extra à plataforma, sendo um participante normal que participa e utiliza os serviços suportados por *JXTA*;
- *Rendezvous*: este modo está associado a um participante que tem recursos extra e pretende disponibilizá-los;
- *Relay*: este modo oferece mecanismos de transmissão de mensagens que diferem dos habitualmente utilizados;
- *Proxy*: este modo oferece reencaminhamento dos serviços *JXTA* para dispositivos móveis baseados em *J2ME*;
- *Super*: este modo conjuga as funcionalidades dos modos *Rendezvous*, *Relay* e *Proxy*.

Após iniciado *JXTA*, este efectua a filiação automática deste participante no grupo global. Este grupo é importante dado que todas as operações sobre grupos dependem deste grupo global, ou seja, a estruturação dos grupos no sistema *JXTA* utiliza este grupo global como raiz de todos os grupos que serão criados, sendo assim possível construir uma árvore que reflecte a relação entre os diversos grupos do sistema.

5.3.1.2 Remoção de um participante na plataforma

É dada a possibilidade a um participante de sair da plataforma a qualquer momento.

Listing 5.4: Cancelamento de um participante na plataforma

```
1 public void Users.unregister() {
2   Platform.start();
3 }
```

Como previamente mencionado, quando se inicia a plataforma sem ter um nome associado, será utilizado o nome *undefined* para definir este participante, sendo esse um nome reservado. Ao reiniciar-se a plataforma, o nome previamente obtido antes da execução da primitiva `Users.unregister` é libertado e a plataforma continua a ser executada sob o nome *undefined* até que seja efectuada a terminação da plataforma. Desta forma, um participante que não se encontre registado é um participante que participa na plataforma mas não de forma explícita, isto é, este participante pode ajudar a encaminhar consultas, pesquisas de anúncios, entre outros, que são tarefas que *JXTA* executa em *background*.

Gestão de grupos e eventos de forma persistente

A classe *Platform* implementa as funções de salvaguarda (`Platform.save`) e carregamento de todos os identificadores de grupo de que um dado participante é membro (`Platform.load`).

O método de salvaguarda de todos os identificadores dos grupos e subgrupos que esse participante entrou previamente, consiste em percorrer a lista de todos os identificadores e guardá-los de forma persistente, sendo que a implementação deste método foi conseguida através da salvaguarda dessa informação em ficheiro.

Para se carregar o último estado salvaguardado dos grupos e subgrupos associados a esse participante, basta percorrer esse ficheiro, que contém a lista com todos os identificadores e entrar explicitamente nesses grupos.

No registo de um participante, verifica-se se já existia alguma versão salvaguardada de grupos e eventos que este participante tivesse associado, sendo que se existir então tenta-se filiar a todos os grupos e subscrever a todos os eventos presentes nesse ficheiro.

Sempre que um participante entra ou sai de um grupo, são salvaguardados todos os identificadores associados aos grupos e eventos de que esse participante é membro ou subscritor.

Actualizar a *cache* interna

A plataforma disponibiliza uma primitiva para actualizar a *cache* local com os anúncios disponíveis em *JXTA*. Existe uma cache por participante e esta cache guarda os anúncios relativos a grupos ou participantes que se encontram ao longo da execução da plataforma, sendo esta actualizada sempre que se manipula informação relativa a grupos ou participantes.

Esta primitiva existe com o intuito de o programador poder actualizar a *cache* quando lhe achar conveniente, possibilitando assim o funcionamento correcto de primitivas que dependam de pesquisas de anúncios que podem não existir num dado momento na *cache* local.

Existem duas primitivas para actualizar a *cache* local: a função `GroupsManager.updateCache()`, que actualiza a *cache* local quanto aos anúncios de grupos que têm o grupo global definido como grupo "pai" e existe a função `JXTAGroup.updateCache()` que actualiza a *cache* local quanto ao anúncios de subgrupos do grupo em que se está a invocar a primitiva. Este último método é útil quando se sabe da existência de um subgrupo associado a um grupo a que o utilizador pertence mas ainda não se conhece o anúncio associado a esse subgrupo.

Listing 5.5: Actualizar a *cache* interna

```
1 public boolean GroupsManager.updateCache(String groupName) {
2   Group group = getGroup(groupName);
3   group.updateCache();
4 }
5
6 private void JXTAGroup.updateCache(name) {
7   getAdvs(DiscoveryService.GROUP, null, null);
8 }
```

5.3.2 Operações sobre grupos

As operações que se podem efectuar sobre grupos são disponibilizadas através da classe *GroupsManager*, a qual define uma estrutura de dados que guarda informação sobre todos os grupos com os quais já houve algum tipo de interacção. É assim possível utilizar esse objecto que representa o grupo quando se pretende efectuar alguma operação sobre esse grupo, evitando estar sempre à procura do grupo quando se pretende efectuar alguma operação sobre esse grupo.

A representação de um grupo nessa estrutura de dados é efectuada através da classe *Group*. Essa classe contém uma instância da classe *JXTAGroup* que reflecte o próprio grupo que esta classe representa. Para além desse objecto, existe uma estrutura de dados que contém todos os subgrupos associados ao grupo que esta classe representa. A classe *Group* disponibiliza todas as primitivas associadas às operações sobre grupos, realizando o seu mapeamento directo sobre a classe *JXTAGroup*.

5.3.2.1 Criação de um grupo

A criação de um grupo na plataforma é efectuada através da primitiva `GroupsManager.create` (listagem de código 5.6) e esta primitiva é dividida nos seguintes passos: primeiro verifica-se se o nome pretendido já se encontra associado a outro grupo (linha 2 e 3) e caso exista, então a criação é abortada e é retornado *false* (linha 4); caso contrário

prossegue-se; de forma a partilhar a existência de um novo grupo é necessário criar um anúncio associado a esse grupo (linha 6) e publicar esse anúncio remotamente (linha 7); de seguida instancia-se um objecto do tipo *PeerGroup* que reflecte este grupo (linha 8) e guarda-se esse grupo na estrutura de dados de grupos; de forma a avisar o gestor do espaço partilhado (participante *sharedspace*), é enviada uma mensagem a mencionar a criação deste grupo (linha 10), de forma a que esse participante crie um espaço partilhado para este novo grupo (esta criação é descrita em pormenor na secção 5.3.4); por fim, é a primitiva retorna *true*.

Listing 5.6: Criação de um grupo

```
1  public boolean GroupsManager.create(name) {
2    PeerGroup res = findGroup(parent, name);
3    if(res!=null)
4      return false;
5    else {
6      adv = GroupsManager.createPeerGroupAdv(parent, name);
7      parent.discoveryService.publish(adv);
8      PeerGroup grp = parent.newGroup(adv);
9      groups.put(name,new Group(grp));
10     Platform.users.send("sharedspace", "create", name, "sharedspace", "");
11     return true;
12   }
13 }
```

De seguida apresentam-se as funções privadas utilizadas na implementação da primitiva `GroupsManager.create`.

Criação de um anúncio associado a um grupo

Relativamente à identificação de um grupo, é preciso ter em atenção que um grupo é identificado através de um anúncio que contém as informações essenciais desse grupo, sendo particularmente importante o identificador de sistema desse grupo. Para tal, antes da criação de um grupo é necessário criar o anúncio que contém as informações essenciais associadas ao grupo. A criação do anúncio passa pela invocação da função privada `GroupsManager.createPeerGroupAdv` presente na plataforma *Imagine*.

Listing 5.7: Criação de um anúncio associado a um grupo

```
1  private PeerGroupAdvertisement GroupsManager.createPeerGroupAdv(parent, name)
2    adv = AdvertisementFactory.newAdvertisement();
3    adv.setPeerGroupID(newPeerGroupID(parent));
4    adv.setName(name);
5    return adv;
6  }
```

Todos os grupos criados e respectivos anúncios derivam sempre de um grupo já previamente estabelecido, que em última instância será o grupo global obtido quando se

inicia a plataforma. A criação do anúncio de um grupo (listagem de código 5.7) é concretizada através da criação de um anúncio genérico (linha 2). Após a criação desse anúncio define-se qual o identificador de sistema que será associado ao grupo (linha 3) e respectivo nome (linha 4). O novo identificador de sistema é gerado a partir do nome do grupo que foi submetido como argumento. O grupo submetido torna-se o grupo "pai" do anúncio que se está a criar.

Aceder ao objecto que representa um grupo

De forma a aceder ao objecto que representa um grupo, é disponiblizada a função privada `GroupsManager.getGroup` que é utilizada internamente na plataforma, na qual é necessário fornecer como argumento qual o nome do grupo. Se já se conhecer o objecto associado a esse nome então é devolvido esse objecto, caso contrário é efectuada uma pesquisa do grupo através da função `GroupsManager.findGroup`.

Listing 5.8: Representação de um grupo na plataforma

```
1 private Group GroupsManager.getGroup(name) {
2   if(groups.containsKey(name))
3     return groups.get(name);
4   else {
5     return GroupsManager.findGroup(Platform.worldGroup, name);
6   }
7 }
```

Pesquisa de um grupo na plataforma

A pesquisa de um grupo funciona de uma de duas formas (listagem de código 5.9): ou se fornece um identificador de sistema utilizado pelo sistema *JXTA* (linha 2 e 3), sendo possível neste caso obter logo o grupo correcto visto os identificadores desses grupos serem únicos; ou se fornece um nome (linha 5 e 6), sendo neste caso necessário invocar a função privada `GroupsManager.getPeerGroupAdv` que procura pelo anúncio que representa o grupo com esse nome. Após invocada essa função cria-se o grupo a partir do identificador de sistema contido no anúncio.

Listing 5.9: Procura de um grupo na plataforma

```
1 private PeerGroup GroupsManager.findGroup(parent, name) {
2   if(name.startsWith("urx:")) {
3     return parent.newGroup(PeerGroupID.create(name));
4   } else {
5     adv = GroupsManager.getPeerGroupAdv(parent, name);
6     return parent.newGroup(adv.getPeerGroupID());
7   }
8 }
```

5.3.2.2 Eliminação de um grupo

Para se eliminar um grupo basta invocar a primitiva `GroupsManager.destroy` que recebe como argumento o nome lógico desse grupo e um booleano que indica se a eliminação deve ser imediata ou não, de acordo com a descrição apresentada no capítulo anterior.

Listing 5.10: Eliminação de um grupo

```
1  public void GroupsManager.destroy(name, now) {
2    getGroup(name).destroy(now);
3  }
4
5  private void JXTAGroup.destroy(now) {
6    if(now) {
7      send("destroy","destroy", "", "");
8    } else {
9      adv = group.getPeerGroupAdvertisement();
10     adv.setDescription("destroy");
11     discoveryService.remotePublish(a);
12   }
13   group.leave();
14   discoveryService.flushAdvertisement();
15 }
```

A primitiva `GroupsManager.destroy` (listagem de código 5.10) elimina um dado grupo e respectivos subgrupos associados a esse grupo, sendo que a eliminação consiste nos seguintes passos: primeiro verifica-se se é uma eliminação imediata; se for, então é gerada uma mensagem a mencionar esta acção para todos os membros do grupo (linha 7); caso contrário, então é alterada a descrição do anúncio referente a este grupo de forma a reflectir a eliminação do grupo (linha 9 a 11); de seguida, é efectuada a saída do grupo (linha 13) e por fim, elimina-se localmente o anúncio associado ao grupo eliminado (linha 14). Dado que a existência de um grupo é alcançada através da distribuição do anúncio associado a esse grupo, é necessário eliminar todos os anúncios referentes a esse grupo de forma a que este deixe de existir na plataforma.

5.3.2.3 Filiação num grupo

A filiação de um participante num grupo é efectuada com a primitiva `GroupsManager.join` (listagem de código 5.11), sendo que essa primitiva conta com os seguintes passos: primeiro é necessário obter o objecto referente a esse grupo, através da função privada `GroupsManager.getGroup` (linha 2); de seguida, é efectuada a filiação nesse grupo através do objecto obtido (linha 3); após efectuada a filiação, são salvaguardados todos os grupos a que um participante pertence de forma persistente, através da primitiva `Platform.save` (linha 4); por fim, é actualizada a *cache* quanto a possíveis anúncios de subgrupos no contexto do grupo em que se acabou de filiar (linha 5).

Listing 5.11: Filiação num grupo

```
1  public void GroupsManager.join(name, credential) {
2    group = GroupsManager.getGroup(name);
3    group.join(credential);
4    Platform.save();
5    group.updateCache();
6  }
7
8  private void JXTAGroup.join(credential) {
9    auth = membershipService.apply(credential);
10   passport = membershipService.join(auth);
11 }
```

Relativamente à filiação num grupo, esta consiste em submeter ao serviço de filiação desse grupo, uma credencial que valide o participante (linha 9). Após efectuada essa validação com sucesso, é gerada uma nova credencial que valida este participante como membro do grupo (linha 10).

As credenciais que se podem submeter aquando da filiação seguem a descrição apresentada no capítulo anterior, na descrição desta primitiva.

5.3.2.4 Saída de um grupo

A saída de um participante de um grupo é efectuada com a primitiva `GroupsManager.leave` (listagem de código 5.12), onde é necessário fornecer como argumento, o nome do grupo de que se pretende sair. A saída de um grupo consiste em invalidar a credencial que valida este participante como membro do grupo (linha 7).

Após efectuada a saída do grupo, é salvaguardada a nova constituição dos grupos ao qual este participante está filiado (linha 3).

Listing 5.12: Saída de um grupo

```
1  public void GroupsManager.leave(name) {
2    GroupsManager.getGroup(name).leave();
3    Platform.save();
4  }
5
6  private void JXTAGroup.leave() {
7    membershipService.resign(passport);
8  }
```

5.3.2.5 Constituição de um grupo

De forma a obter todos os membros de um certo grupo, é disponibilizada a primitiva `GroupsManager.membership` (listagem de código 5.13). Esta primitiva consiste em obter todos os anúncios, através da função privada `JXTAGroup.getAdvs`, associados a membros deste grupo, sendo que o nome dos membros está contido no anúncio. A

invocação dessa função tem como argumento a obtenção de anúncios apenas referentes a participantes e não aplicando qualquer restrição de pesquisa.

Listing 5.13: Constituição de um grupo

```
1 public List<String> GroupsManager.membership(name) {
2   return GroupsManager.getGroup(name).getAdvs(PEER, null, null).toString();
3 }
```

5.3.3 Gestão de comunicação

Esta secção apresenta as primitivas associadas à comunicação directa entre participantes (*1-1*) e difusão de mensagens entre membros de um mesmo grupo (*1-n*). Como mencionado no capítulo anterior, a comunicação baseia-se na troca de mensagens, em que o envio de mensagens é efectuado explicitamente através de uma primitiva e a recepção de mensagens é efectuada através de objectos de escuta (*listeners*).

Quanto às mensagens trocadas na plataforma, estas contêm uma lista de pares em que o primeiro valor reflecte o tipo de conteúdo e o segundo valor contém o respectivo conteúdo, sendo a estrutura destas mensagens descrita em pormenor na secção 4.4.3.

5.3.3.1 Comunicação directa

Relativamente à comunicação directa entre dois participantes, o envio de mensagens é dividido em duas primitivas, em que a primeira primitiva permite o envio de uma mensagem com apenas um par de conteúdo, não sendo assim necessário criar explicitamente a lista de conteúdos a enviar, e uma segunda primitiva que envia uma lista de conteúdos que segue a estrutura apresentada na secção 4.4.3.

Listing 5.14: Envio de uma mensagem para um participante

```
1  public void Users.send(name, type, message, listenerName, listenerParam) {
2    list = <type, message>;
3    Users.send(name, list, listenerName, listenerParam);
4  }
5
6  public void Users.send(name, list, listenerName, listenerParam) {
7    adv = Platform.worldGroup.getAdvs(DiscoveryService.PEER, "Name", name);
8    m = Platform.worldGroup.endpointService.getMessenger(adv);
9    m.waitState(RESOLVED, 5000);
10   m.sendMessage(msg, listenerName, listenerParam);
11 }
```

Em ambas as primitivas (listagem de código 5.14), o envio da mensagem é efectuado da mesma forma e esse envio conta com os seguintes passos: inicialmente é necessário obter o anúncio relativo ao participante para o qual se pretende enviar a mensagem (linha 7); após a obtenção desse anúncio, é criado um mensageiro (*Messenger*) (linha 8) que abre um canal de comunicação entre os dois participantes; com o canal de comunicação criado,

é necessário verificar se o canal se encontra disponível e apto a receber novas mensagens (linha 9), se estiver então é enviada a mensagem para o objecto de escuta definido como argumento (linha 10), caso contrário é bloqueado o processamento desta função até que o canal esteja pronto. É possível definir um tempo máximo de espera, de forma a não ficar bloqueado indefinidamente nesta função, sendo que esta funcionalidade é importante caso um certo participante já não esteja disponível na plataforma.

A classe *Messenger* é uma classe disponibilizada por *JXTA* que permite a comunicação entre dois participantes.

5.3.3.2 Difusão de mensagens por membros de um grupo

A difusão de mensagens por membros de um grupo segue a mesma abordagem inerente ao envio de mensagens directo entre participantes, isto é, existem duas primitivas para a difusão de mensagens por membros de um grupo, em que a primeira permite o envio de apenas um par de conteúdo e tipo associado e a segunda primitiva permite o envio de uma lista de conteúdos.

Listing 5.15: Difusão de uma mensagem para um grupo

```
1  public void GroupsManager.send(name, type, message, listenerName, listenerParam
       ) {
2    list = <type, message>;
3    send(name, list, listenerName, listenerParam);
4  }
5
6  public void GroupsManager.send(name, list, serviceName, serviceParam) {
7    group = GroupsManager.getGroup(name);
8    if(group.checkDestroy())
9      return;
10   else
11     group.endpointService.propagate(list, serviceName, serviceParam);
12 }
13
14 private boolean JXTAGroup.checkDestroy() {
15   PeerGroupAdvertisement pga = JXTAGroup.getAdvs(GROUP, "Name", group.
         getPeerGroupName()).nextElement();
16   if(pga==null || pga.getDescription()!=null && pga.getDescription().
         equalsIgnoreCase("destroy")) {
17     JXTAGroup.destroy(false);
18     return true;
19   }
20   return false;
21 }
```

Em relação às duas primitivas disponíveis para a difusão de mensagens por membros de um grupo (listagem de código 5.15), ambas seguem o mesmo conjunto de passos: antes de se tentar difundir uma mensagem para um grupo, é verificado se esse grupo foi marcado para eliminação (linha 8). Essa verificação consiste em obter remotamente

o anúncio associado ao grupo, através da função privada `JXTAGroups.getAdvs` (linha 15) e verificar se este tem indicação de que o grupo foi marcado para eliminação, ou caso o anúncio não exista (linha 16) então procede-se à eliminação não imediata do grupo (linha 17); caso o grupo não tenha sido marcado para eliminação, a difusão da mensagem é efectuada através do serviço de comunicação de mais baixo nível associado a esse grupo, o *EndpointService* (linha 11). A difusão da mensagem consiste em enviar uma lista de conteúdos para o objecto de escuta definido como argumento.

5.3.3.3 Recepção de mensagens

Como mencionado, a recepção de mensagens na plataforma *Imagine* é efectuada à custa de objectos de escuta e é necessário registar esses objectos de escuta por cada participante, no contexto de um grupo com um determinado nome e parâmetro.

Listing 5.16: Registo de um objecto de escuta

```
1 public void GroupsManager.addIncomingListener(groupName, listener, listenerName
      , listenerParam) {
2   if(groupName!=null)
3     GroupsManager.getGroup(groupName).endpointService.addIncomingListener(
          listener,listenerName, listenerParam);
4   else
5     Platform.worldGroup.endpointService.addIncomingListener(listener,
          listenerName, listenerParam);
6 }
```

A primitiva `GroupsManager.addIncomingListener` (listagem de código 5.16) permite registar objectos de escuta no contexto de um grupo, com um determinado nome e parâmetro. Caso não seja submetido um nome de grupo então o grupo ao qual se regista este objecto de escuta é o grupo global.

Também é possível remover objectos de escuta previamente registados, através da primitiva `GroupsManager.removeIncomingListener` (listagem de código 5.17. Esta remoção do objecto de escuta consiste em remover a instância do objecto, bem como libertar o nome e parâmetro do objecto de escuta previamente registado.

Listing 5.17: Eliminação de um objecto de escuta

```
1 public void removeIncomingListener(groupName, listenerName, listenerParam) {
2   if(groupName!=null && !groupName.equals(""))
3     GroupsManager.getGroup(groupName).endpointService.removeIncomingListener(
          listenerName, listenerParam);
4   else
5     Platform.worldGroup.endpointService.removeIncomingListener(listenerName,
          listenerParam);
6 }
```

5.3.4 Gestão do espaço partilhado

Como mencionado na secção 5.2 deste capítulo, o espaço partilhado é implementado em dois contextos, o de cliente e o do servidor.

5.3.4.1 Cliente

Do lado do cliente, é possível efectuar operações sobre o espaço partilhado de um grupo através da primitiva `GroupsManager.sharedSpace`. A comunicação com o espaço partilhado de um grupo é efectuada através do envio de mensagens directas para o participante *sharedspace*. As respostas obtidas após concluída a operação sobre o espaço partilhado, são enviadas para o objecto de escuta registado com o nome *sharedspace* e o parâmetro com o valor submetido no argumento *replyTo*.

Listing 5.18: Manipulação de um espaço partilhado

```
1 public void GroupsManager.sharedSpace(groupName, command, numOfFields, fields,
      wait, replyTo) {
2   tuple = new Tuple(numOfFields, fields);
3   list = <tuple, command, wait, replyto>;
4   Platform.Users.send("sharedspace", list, "sharedspace", groupName);
5 }
```

A implementação da primitiva `GroupsManager.sharedSpace` (listagem de código 5.18), passa por criar um tuplo com os argumentos submetidos (linha 2), criar uma lista com esse tuplo e os restantes argumentos submetidos como argumento (linha 3) e enviar essa lista de conteúdo para o gestor do espaço partilhado, o participante *sharedspace*.

5.3.4.2 Servidor

Quanto à implementação do lado do servidor, esta consiste numa classe *Server* que inicia um participante em *JXTA* com o nome *sharedspace*. Esse participante regista um objecto de escuta para receber os pedidos de criação de espaços partilhados, pedidos estes que são submetidos quando um grupo é criado (listagem de código 5.19, linha 4). Sempre que um espaço partilhado é criado para um grupo, é verificado se existe alguma versão desse espaço partilhado salvaguardada de forma persistente e, se existir, então essa versão é carregada.

Listing 5.19: Manipulação de um espaço partilhado

```
1 public class ServerEndpointListener extends SharedSpaceEndpointListener {
2   public void processIncomingMessage(Message msg, EndpointAddress srcAddr,
        EndpointAddress dstAddr) {
3     if(msg.getMessageElement("create")!=null)
4       addSharedSpaceManagerListener(msg.getMessageElement("create"));
5     else
6       super.processIncomingMessage(msg, srcAddr, dstAddr);
7   }
8
```

```
9     private void addSharedSpaceManagerListener(String groupName) {
10      netPeerGroup.addIncomingMessageListener(new SharedSpaceEndpointListener(
            netPeerGroup, groupName), "sharedspace", groupName);
11    }
12  }
```

Os restantes pedidos submetidos a este participante são operações que se devem executar sobre o espaço partilhado de um determinado grupo, sendo essas operações reencaminhadas (linha 6) para um objecto de escuta que faz o tratamento dessas operações (listagem de código 5.20). Todas as operações implementadas seguem a lógica descrita anteriormente na secção 4.4.4.

Listing 5.20: Manipulação de um espaço partilhado

```
1  public class SharedSpaceManager {
2    private List<Tuple> tuples;
3    private List<LockedTuple> locks;
4
5    public Tuple get(Tuple t, boolean wait) {
6      int ind;
7      synchronized(tuples) {
8        ind = tuples.indexOf(t);
9        if(ind!=-1)
10         return tuples.remove(ind);
11     }
12     while(ind==-1 && wait) {
13       Object lock = new Object();
14       LockedTuple lt = new LockedTuple(t, lock);
15       synchronized(locks) {
16         locks.add(lt);
17       }
18       synchronized(lock) {
19         lock.wait();
20       }
21       synchronized(tuples) {
22         ind = tuples.indexOf(t);
23         if(ind==-1)
24           continue;
25         return tuples.remove(ind);
26       }
27     }
28     return null;
29   }
30 }
```

Em relação aos pedidos que são bloqueantes, estes são resolvidos recorrendo a um objecto genérico que bloqueia até que seja detectado um tuplo que satisfaça a operação pretendida.

Por exemplo, se um participante quiser remover de um espaço partilhado vazio, um tuplo com a estrutura <_>, ou seja, um tuplo com apenas um campo e que contenha

qualquer valor nesse campo, esta operação só será concluída com sucesso quando for introduzido um tuplo com apenas um campo.

O bloqueamento do fluxo de execução desta operação é alcançado através da criação de um objecto genérico (linha 13). Esse objecto genérico é associado ao tuplo que se procura (linha 14) e é efectuada uma espera até que outro processo manipule este objecto genérico (linha 18 a 20).

Todas as operações sobre o espaço partilhado são operações atómicas. Em caso de haver vários participantes que pretendem efectuar operações sobre o mesmo tuplo de um espaço partilhado, o desempate dessas operações é efectuado pelo servidor através da utilização de blocos de código sincronizados para acesso aos tuplos de um espaço partilhado (linha 21 à 26).

5.3.5 Gestão de eventos

A implementação do sistema de eventos na plataforma é alcançada através da criação de subgrupos para cada evento criado. Visto o sistema de eventos seguir o modelo de publicação/subscrição, no qual existem emissores activos e receptores que registam o interesse em receber mensagens de um determinado evento, a abordagem lógica para implementar este modelo sobre o sistema *JXTA*, recorreu à utilização do modelo de grupos oferecidos pelo sistema. Tendo esta premissa em conta, todas as operações possíveis sobre o sistema de eventos têm mapeamento directo com operações que se efectuam sobre grupos.

5.3.5.1 Criação de um evento

A criação de um evento é efectuada através da primitiva `GroupsManager.advertise` (listagem de código 5.21), na qual se fornece como argumento o nome do grupo e o nome do evento que se pretende criar. Esta primitiva consiste em criar um subgrupo com o nome do evento associado ao nome do grupo fornecido, sendo essa criação de grupo alcançada através da primitiva `GroupsManager.create` e o valor retornado por essa primitiva é devolvido como resultado da execução da criação de um evento.

Listing 5.21: Criação de um evento

```
1 public boolean GroupsManager.advertise(groupName, eventName) {
2   return GroupsManager.create(groupName, eventName);
3 }
```

5.3.5.2 Eliminação de um evento

É possível a qualquer membro de um grupo eliminar esse grupo através da primitiva `GroupsManager.unadvertise` (listagem de código 5.22), fornecendo o nome do grupo e nome do evento que se pretende eliminar. A eliminação de um evento consiste na eliminação imediata de um subgrupo, sendo essa eliminação alcançada a partir da primitiva `GroupsManager.destroy`. Como referido na descrição dessa primitiva, a eliminação

imediata consiste em enviar uma mensagem a todos os membros desse grupo com a indicação da eliminação deste subgrupo.

Listing 5.22: Eliminação de um evento

```
1 public void GroupsManager.unadvertise(groupName, eventName) {
2   GroupsManager.getGroup(groupName).getEvent(eventName).destroy(true);
3 }
```

5.3.5.3 Subscrição de um evento

A subscrição a um evento é efectuada através da primitiva `GroupsManager.subscribe` (listagem de código 5.23), na qual se fornece como argumento o nome do grupo e o nome do evento que se pretende subscrever. Esta primitiva consiste em efectuar a filiação ao subgrupo com o nome do evento associado ao nome do grupo fornecido.

Listing 5.23: Subscrição de um evento

```
1 public void GroupsManager.subscribe(groupName, eventName) {
2   GroupsManager.getGroup(groupName).getEvent(eventName).join();
3 }
```

5.3.5.4 Anular a subscrição de um evento

Relativamente à anulação da subscrição de um evento, esta é alcançada através da primitiva `GroupsManager.unsubscribe` (listagem de código 5.24), em que é necessário fornecer como argumento o nome do grupo e o nome do evento ao qual se pretende anular a subscrição. Esta primitiva consiste em efectuar a saída de um subgrupo com o nome do evento associado ao nome do grupo fornecido.

Listing 5.24: Anular a subscrição de um evento

```
1 public void GroupsManager.unsubscribe(groupName, eventName) {
2   GroupsManager.getGroup(groupName).getEvent(eventName).leave();
3 }
```

5.3.5.5 Envio de uma mensagem associada a um evento

A difusão (publicação) de mensagens relativas a um evento é efectuada com a primitiva `GroupsManager.publish` (listagem de código 5.25) que homologamente às primitivas de difusão de mensagens por membros de um grupo, também se divide em duas. Existe uma primitiva para o envio de uma mensagem com apenas um par de conteúdo e outra primitiva para o envio de uma lista de conteúdos. A difusão de mensagens associadas a um evento é alcançada utilizando a primitiva `GroupsManager.send` que permite a difusão de mensagens por membros de um mesmo grupo.

Listing 5.25: Publicação de uma mensagem associada a um evento

```
1 public void GroupsManager.publish(groupName, type, message, eventName,
      listenerParam) {
2   list = <type, message>;
3   publish(groupName, list, eventName, eventParam);
4 }
5
6 public void GroupsManager.publish(groupName, list, eventName, listenerParam) {
7   GroupsManager.getGroup(groupName).getEvent(eventName).send(list, eventName,
        listenerParam);
8 }
```

5.3.5.6 Recepção de mensagens associada a um evento

Relativamente à recepção de mensagens relativas a um evento, esta é alcançada à custa de um objecto de escuta, à semelhança da difusão de mensagens por membros de um grupo. Para tal, é disponibilizada a primitiva `GroupsManager.addIncomingEventListener` (listagem de código 5.26), que permite o registo do objecto de escuta fornecido como argumento no subgrupo que representa o evento, sendo o objecto de escuta registado com o nome do evento e parâmetro fornecido como argumento.

Listing 5.26: Registo de um objecto de escuta associado a um evento

```
1 public void GroupsManager.addIncomingEventListener(groupName, listener,
      eventName, listenerParam) {
2   if(groupName!=null)
3     GroupsManager.getGroup(groupName).getEvent(eventName).addIncomingListener(
          listener, eventName, listenerParam);
4 }
```

Por fim e à semelhança da manipulação de objectos de escuta associados a grupos, é possível remover objectos de escuta previamente registados a um evento através da primitiva `GroupsManager.removeIncomingEventListener` (listagem de código 5.27).

Listing 5.27: Eliminação de um objecto de escuta associado a um evento

```
1 public void GroupsManager.removeIncomingEventListener(groupName, eventName,
      eventParam) {
2   if(groupName!=null && !groupName.equals(""))
3     getGroup(groupName).getEvent(eventName).removeIncomingListener(eventName,
          eventParam);
4 }
```

5.3.6 Mapeamento das funcionalidades suportadas sobre a plataforma JXTA

Esta secção resume o mapeamento das funcionalidades suportadas sobre as classes disponíveis em *JXTA*. Esse resumo é efectuado através de um conjunto de tabelas (tabela 5.1, 5.2, 5.3 e 5.4), separadas pelos componentes principais presentes na plataforma *Imagine*.

Plataforma *Imagine*	Classes utilizadas de *JXTA*
`GroupsManager.create`	DiscoveryService PeerGroup Advertisement
`GroupsManager.destroy`	DiscoveryService PeerGroup Advertisement
`GroupsManager.membership`	MembershipService PeerGroup AuthenticationCredential Authenticator
`GroupsManager.leave`	MembershipService PeerGroup
`GroupsManager.membership`	DiscoveryService PeerGroup Advertisement

Tabela 5.1: Funcionalidades suportadas de gestão de grupos sobre *JXTA*

Plataforma *Imagine*	Classes utilizadas de *JXTA*
`Users.send`	EndpointService Advertisement Messenger Message
`GroupsManager.send`	EndpointService Message
`GroupsManager.addIncomingListener`	EndpointService EndpointListener
`GroupsManager.removeIncomingListener`	EndpointService

Tabela 5.2: Funcionalidades suportadas de comunicação entre participantes sobre *JXTA*

Plataforma *Imagine*	Classes utilizadas de *JXTA*
`GroupsManager.sharedSpace`	EndpointService EndpointListener Message

Tabela 5.3: Funcionalidades suportadas de gestão do espaço partilhado sobre *JXTA*

5.4 Conclusão

Este capítulo descreveu os detalhes de implementação da plataforma *Imagine*, tendo sido descritas as primitivas disponibilizadas por esta, bem como algumas funções auxiliares relevantes ao funcionamento da plataforma.

Todas as funcionalidades apresentadas e respectiva primitivas, encontram-se implementadas sobre a forma de um protótipo.

Plataforma *Imagine*	Classes utilizadas de *JXTA*
`GroupsManager.advertise`	DiscoveryService PeerGroup Advertisement
`GroupsManager.unadvertise`	DiscoveryService PeerGroup Advertisement
`GroupsManager.subscribe`	MembershipService PeerGroup AuthenticationCredential Authenticator
`GroupsManager.unsubscribe`	MembershipService PeerGroup
`GroupsManager.publish`	EndpointService Message
`GroupsManager.addIncomingEventListener`	EndpointService EndpointListener
`GroupsManager.removeIncomingEventListener`	EndpointService

Tabela 5.4: Funcionalidades suportadas de gestão de eventos sobre *JXTA*

Após apresentada a plataforma e seus detalhes de implementação, é necessário validar e exemplificar a utilização da mesma. Para tal, foi implementado um jogo colaborativo que tira partido das primitivas oferecidas pela plataforma como forma de estruturar a organização e as interacções nos diferentes cenários de colaboração presentes no jogo.

6

Um Jogo Colaborativo

Este capítulo descreve um jogo colaborativo desenvolvido sobre a plataforma *Imagine*. O desenvolvimento deste jogo teve como principal objectivo validar a plataforma *Imagine*, utilizando as primitivas por esta disponibilizadas na resolução dos diferentes cenários de colaboração e interacção presentes no jogo.

O capítulo começa por fazer considerações gerais que levaram a escolher um jogo colaborativo como exemplo de aplicação de validação da plataforma desenvolvida. De seguida é descrito o jogo, sendo ilustrados alguns cenários básicos de funcionamento. Por fim, é detalhada a implementação das diferentes fases do jogo sobre a plataforma *Imagine*.

6.1 Considerações Gerais

Existiam diversos cenários de colaboração possíveis para validar a plataforma *Imagine*, desde cenários de colaboração educativos, em que se incentiva à aprendizagem em grupo, aplicações de edição de documentos de texto a partir de diversos participantes, ou cenários de colaboração mais lúdicos, por exemplo jogos em que se incentiva à colaboração.

O processo inicial de criação de um jogo costuma ser complexo, dado que este requer que se definam correctamente os aspectos que definem todo o comportamento base do jogo, isto é, a lógica básica do jogo. Após esta fase inicial, a extensão do jogo é mais simples que o processo inicial, dependendo apenas da imaginação dos criadores do jogo, como por exemplo, a criação de novos mapas, ou a adição de novas funcionalidades.

Com a implementação deste jogo e tendo em conta os diversos cenários de colaboração que foram implementados, é possível testar e validar a plataforma *Imagine*, tirando partindo do motor gráfico 2D *JGame* para a construção de uma interface mais interessante

para o jogo.

Quanto à validação de jogadas e possíveis ilegalidades cometidas pelos jogadores, esta problemática é resolvida de maneira idêntica aos jogos multi-jogador centralizados, isto é, as mensagens trocadas devem utilizar um mecanismo que impossibilite a alteração do conteúdo da mensagem e todas as mensagens devem ser validadas do lado do jogador.

Por exemplo, num jogo em que existe um mapa que é representado por uma quadrícula de cem por cem e em que cada jogador só pode mover um quadrado por ronda, se houver um jogador que indica aos restantes jogadores que deseja mover mais do que um quadrado, é necessário que os restantes jogadores não aceitem essa jogada, requerendo assim mais processamento do lado de cada jogador mas evitando comunicação constante com o servidor.

Por fim, quanto ao desenvolvimento de um jogo colaborativo, há que definir a lógica básica e quais os mecanismos de interacção entre os diversos jogadores, não sendo necessário ponderar em concreto todas as possibilidades de colaboração existentes para atingir um determinado objectivo. Apenas se fornecem os mecanismos de colaboração e cabe a cada jogador tirar proveito desses mecanismos para atingir o objectivo do jogo.

6.2 Especificação do Jogo

O jogo é jogado por uma ou mais equipas de jogadores, em que cada equipa tem como objectivo coleccionar o maior número de tesouros presentes num campo do jogo no menor intervalo de tempo. A eficácia de cada equipa prende-se com a forma como a comunicação e colaboração são estabelecidas entre os seus membros, de modo a ser recolhido o maior número de objectos no menor intervalo de tempo.

Cada jogador tem uma versão do jogo no dispositivo que o executa, ficando assim com ficheiros comuns entre os diferentes jogadores.

6.2.1 Campo do jogo

Uma instância do jogo tem um campo do jogo no qual são inseridas uma ou mais equipas. Esse campo é constituído por uma grelha de quarenta por quarenta unidades e por um conjunto de obstáculos que limitam a movimentação dos jogadores. Nesse campo existe um conjunto de tesouros que podem ser capturados. A configuração do campo de jogo é pré-definida, estática e encontra-se descrita em ficheiros locais a todos os jogadores, sendo apenas necessário indicar qual o campo de jogo em que se pretende jogar.

A descrição de um campo do jogo é efectuada através de um ficheiro de texto, no qual cada linha representa um elemento presente no campo. Uma linha é constituída por um carácter inicial que define qual o tipo de elemento, seguida da definição das coordenadas no eixo dos xx e das coordenadas no eixo dos yy.

É possível definir os seguintes tipos de elementos num campo do jogo:

- *Ponto de inserção* (**s**): este elemento representa um ponto onde os jogadores de uma equipa são inseridos no início de um jogo (*spawn point*). Este ponto de inserção é importante na definição de um campo do jogo e define quantas equipas é possível ter a jogar num mesmo campo, sendo que o número de pontos de inserção corresponde ao número de equipas possíveis.

- *Tesouros* (**t**): este elemento representa um tesouro num determinado par de coordenadas, sendo que este tesouro ocupa uma unidade de largura e de altura. A nomeação dos tesouros é automática, sendo que os tesouros têm o nome "treasureX", em que X é o número do tesouro, por ordem de ocorrência no ficheiro;

- *Obstáculos*: na actual implementação do jogo, é possível definir dois tipos de obstáculos:

  - *Árvore* (**o**): este elemento representa uma árvore num determinado par de coordenadas, sendo que este obstáculo tem um tamanho pré-definido de uma unidade de largura por duas unidades de altura.
  - *Montanha* (**m**): este elemento representa uma montanha num determinado par de coordenadas, de forma a definir as dimensões da montanha, é necessário introduzir dois valores adicionais no final da linha, por exemplo, "m 1 1 2 2", corresponde a uma montanha nas coordenadas (1,1) com largura de dois e altura de dois. Uma montanha tem de ter sempre no mínimo duas unidades de largura e uma unidade de altura;

Relativamente aos tesouros presentes num campo do jogo, esses têm as seguintes propriedades:

- Iniciação: são adicionados no início do jogo ou podem ir surgindo de forma dinâmica no jogo;

- Quantidade: existe um número constante de tesouros ou podem surgir novos tesouros com o decorrer do jogo;

- Movimentação: podem ter posição fixa ou podem deslocar-se pelo campo do jogo.

No caso particular da configuração ensaiada do jogo, os tesouros são de posição fixa, definidos no início do jogo (isto é, antes de os jogadores conseguirem tomar alguma acção) e em número fixo.

Os restantes elementos (montanhas, árvores e pontos de inserção) são estáticos e em número fixo, de acordo com o ficheiro de descrição do campo do jogo.

6.2.1.1 Exemplo de um campo do jogo

Segue-se o exemplo da definição de um campo do jogo (listagem de código 6.1) e respectiva apresentação no motor gráfico (figura 6.1).

Este pequeno exemplo define um campo do jogo com quatro montanhas, três árvores, dois tesouros e um ponto de inserção.

Listing 6.1: Exemplo de um campo do jogo

```
1  m 6 4 6 8
2  m 26  0 6 4
3  m 36  4 4 4
4  m 18  6 10  4
5  t 24  0
6  t 38  0
7  t 14  2
8  o 14  6
9  o 32  10
10 s 20  2
```

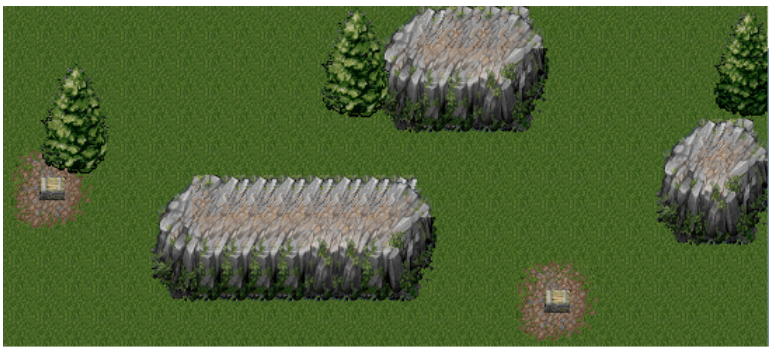

Figura 6.1: Exemplo de um campo do jogo

6.2.2 Jogadores

Um jogador é representado por uma personagem e cada jogador conta com apenas uma habilidade que dá características especiais à sua personagem, sendo com a conjugação das habilidades dos personagens de cada equipa que se consegue optimizar o cumprimento do objectivo do jogo. Actualmente estão implementadas duas habilidades:

- o colector é o jogador que consegue capturar os tesouros que estão presentes no campo do jogo. O colector tem apenas acesso a uma visão parcial do campo, à medida que o jogador se desloca no campo, vão sendo revelados mais detalhes sobre o campo e sobre a posição onde se encontra;
- o observador é um jogador que consegue ver todo o campo do jogo, sendo responsável por fornecer instruções aos colectores da sua equipa de forma a que estes consigam capturar os tesouros. As instruções são transmitidas pelo observador para os colectores da equipa, através de mensagens curtas, ou seja, é imposto um limite de 40 caracteres por mensagem. Note-se que a limitação no tamanho das mensagens obriga a que os observadores tenham de comunicar de forma sucinta e objectiva.

6.2.3 Equipas

Relativamente à organização dos jogadores numa instância do jogo, tem de existir pelo menos uma equipa e cada equipa tem de ter no mínimo um colector. Cada instância suporta até ao máximo de doze jogadores, sendo possível conjugar colectores e observadores por diversas equipas até ao máximo de doze jogadores. Como exemplo de configuração de uma instância, é possível ter uma instância do jogo com duas equipas, em que cada equipa tem dois colectores e um observador, sendo que essa instância suporta seis jogadores.

De forma a incentivar a competição, é possível colocar diversas equipas num mesmo campo do jogo. O objectivo de cada equipa continua a ser o mesmo, isto é, capturar o maior número de tesouros. Os tesouros são comuns a todas as equipas e quando uma equipa captura um tesouro, esse tesouro é removido do campo. De forma a que haja sempre um vencedor em todos os jogos, em caso de empate quanto à captura de tesouros, o desempate consiste em verificar qual foi a equipa que capturou o maior número de tesouros em primeiro lugar.

6.2.4 Estados de uma instância de um jogo

Uma instância de um jogo tem quatro estados possíveis, definidos esquematicamente na figura 6.2.

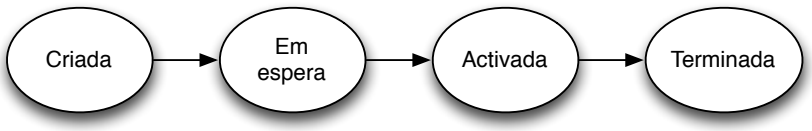

Figura 6.2: Estados de uma instância de um jogo

Uma instância de um jogo tem de ser inicialmente criada por um jogador.

A criação de uma instância consiste em configurar os detalhes dessa instância. É possível configurar os seguintes detalhes:

- nome da instância;
- nome do campo de jogo, que corresponde ao nome do ficheiro que representa o campo;
- número de equipas presentes nesta instância;
- número de observadores por equipa;
- número de colectores por equipa.

Após criada e quando a instância tem pelo menos um jogador, essa instância encontra-se em espera até que as condições de arranque do jogo sejam atingidas. As condições de

arranque consistem em ter todas as equipas e habilidades que se definiram para esta instância preenchidas por jogadores e que todos os jogadores se encontrem preparados para começar o jogo. Esta última indicação é efectuada explicitamente por cada jogador.

Quando essas condições forem atingidas, então o jogo é activado, sendo enviada uma notificação a todos os jogadores. Com a instância do jogo activa, é possível aos jogadores efectuarem todas as acções possíveis dentro do campo do jogo.

Por fim, o jogo termina de acordo com as condições de fim de jogo e esta instância do jogo é destruída. As condições de término de um jogo consistem na captura de todos os tesouros presentes no campo do jogo.

6.2.5 Estados de um jogador

É possível definir um jogador através de um conjunto de estados, sendo possível identificar quatro estados, ilustrados na figura 6.3.

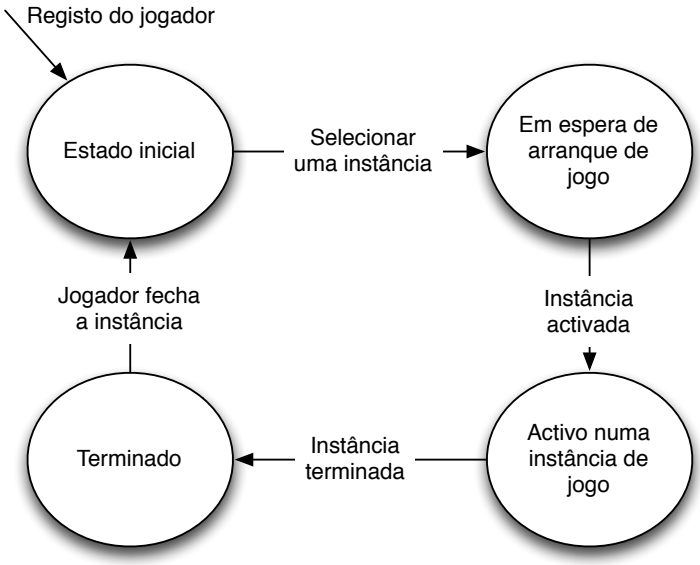

Figura 6.3: Estados de um jogador

Um jogador antes de efectuar qualquer tipo de acção, necessita de se registar na plataforma, ficando com um nome único associado.

Após efectuado o registo, o jogador é encaminhado para o estado inicial, no qual consegue visualizar todas as instâncias de jogos que estão em espera. Neste estado inicial é permitido ao jogador criar uma instância de um jogo ou entrar numa instância já previamente criada.

Se o jogador decidir criar uma instância, então após concluída a configuração dessa instância, esse jogador entra nessa instância e passa ao estado seguinte. Na actual modelação deste jogo colaborativo, só é permitido aos jogadores entrar numa instância de jogo de cada vez.

Quando um jogador entra numa instância, esse jogador encontra-se no estado em espera, isto é, o jogador fica à espera que a instância do jogo seja activada. Neste estado, é possível ao jogador escolher a equipa e habilidade que vai ter nesta instância. Com a equipa e habilidade escolhidas, o jogador deve colocar-se como preparado para arrancar o jogo. Quando todos os jogadores estiverem preparados, a instância do jogo é activada e o jogador passa ao estado seguinte.

Após activada uma instância, o jogador pode efectuar qualquer acção dentro do jogo. Caso seja um observador, o jogador pode mover-se e enviar instruções aos restantes jogadores da sua equipa. Caso seja um colector, então o jogador pode explorar o campo do jogo, movendo-se ao longo do campo e pode capturar os tesouros presentes no campo, caminhando para cima desses tesouros.

Quando todos os tesouros forem capturados, o jogo termina e é mostrada qual a equipa vencedora desta instância, de acordo com as regras de vitória previamente definidas.

O jogador transita para o estado inicial quando fecha esta instância do jogo, isto é, quando o jogador fechar a janela que contém a execução do motor gráfico.

6.3 Implementação do Jogo sobre a Plataforma *Imagine*

Esta secção apresenta a implementação dos diferentes estados presentes numa instância de um jogo, bem como a implementação dos diferentes estados de um jogador e respectiva interface associada a cada estado.

6.3.1 Estados de uma instância de um jogo

De seguida, apresenta-se a implementação efectuada para modelar cada estado associado a uma instância de um jogo e respectivas transições de estado.

6.3.1.1 Criação de uma instância

Quando um jogador cria uma instância do jogo, são criados os grupos necessários ao funcionamento dessa instância de acordo com a configuração pretendida (listagem de código 6.2). Assumindo uma instância com duas equipas serão criados três grupos, um grupo global que reflecte essa instância e no qual os jogadores desse jogo são membros e um grupo para cada equipa.

Listing 6.2: Criação dos grupos necessários numa instância do jogo

```
1 Platform.Groups.create("example");
2 Platform.Groups.create("example_team1");
3 Platform.Groups.create("example_team2");
```

No espaço partilhado do grupo global desse jogo serão colocados os tesouros e um tuplo que reflecte quais os jogadores que se encontram prontos a começar o jogo (listagem de código 6.3).

Cada tesouro é representado no espaço partilhado por um tuplo com um campo, com o valor "treasureX", em que X é o número do tesouro. O tuplo que reflecte quais os jogadores que se encontram prontos é constituído por dois campos, em que o primeiro campo contém o valor "*ready*" e o segundo contém uma lista vazia, visto o jogo ter sido criado nesta fase, que irá conter o nome dos jogadores que já se encontram prontos. A manipulação do tuplo do espaço partilhado associado aos jogadores que se encontram prontos é explicada mais à frente.

Listing 6.3: Preparação do espaço partilhado na criação de uma instância do jogo

```
1 for(int x=0; x<treasure; x++) {
2   fields = <"treasure"+x>;
3   Platform.Groups.sharedspace("example", "update", 1, fields, false, "noreply")
        ;
4 }
5 fields = <"ready", "">;
6 Platform.Groups.sharedSpace("example", "update", 2, fields, false, "noreply");
```

De forma a que os outros jogadores tomem conhecimento desta instância, é necessário registar, no espaço partilhado do grupo global, que existe um novo jogo (listagem de código 6.4). Essa indicação é realizada através de um tuplo com seis campos, em que cada campo tem o seguinte significado: nome do jogo, qual o nome do campo do jogo em que se vai jogar, número de equipas, número de observadores por equipa, número de colectores por equipa e qual o número de jogadores actualmente na instância.

Listing 6.4: Publicação de uma instância do jogo no espaço partilhado do grupo global

```
1 fields = <"example", "level\_1", 2, 2, 1, 0>;
2 Platform.Groups.sharedSpace("", "update", 6, fields, false, "noreply");
```

6.3.1.2 Em espera

Uma instância encontra-se em espera enquanto as condições de arranque não forem respeitadas.

A instância só transita de estado quando um dos jogadores detecta que todos os jogadores se encontram preparados para começar (listagem de código 6.5). Para tal, quando um jogador se coloca como preparado, este tem de remover o tuplo que contém a lista de jogadores que estão preparados. Ao obter essa lista, verifica se os restantes jogadores da instância estão preparados e caso estejam, então significa que todos os jogadores estão prontos a iniciar o jogo e a instância deve ser activada (linha 4 e 5); caso contrário, continua em espera (linha 7 e 8).

Listing 6.5: Verificação das condições de arranque

```
1 class ClientReadyStateSharedSpaceListener implements EndpointListener {
2   public void processIncomingMessage(Message msg, EndpointAddress srcAddr,
        EndpointAddress dstAddr) {
3     String whosReady = msg.getMessageElement("field1");
```

```
4       if(whosReady.split("\t").length==participants.length) {
5         Game.sendStartGame();
6       } else {
7         fields = <"ready", whosReady+"\t"+platform.name};
8         Platform.Groups.sharedSpace("example", "update", 2, fields, true, "
              noreply");
9       }
10    }
11  }
12
13  fields = <"ready", "_">;
14  Platform.Groups.sharedSpace("example", "get", 2, fields, true, "readystate");
```

6.3.1.3 Activação de uma instância

O processo de activação de uma instância consiste nos seguintes passos (listagem de código 6.6): remover a indicação de que o jogo está disponível para novos jogadores do espaço partilhado do grupo global e enviar uma mensagem a todos os jogadores do jogo "example" de que é possível começar o jogo.

Listing 6.6: Activar uma instância do jogo

```
1  void Game.sendStartGame() {
2    list = <"start" -> "example">;
3    Platform.Groups.send("example", list, "game", "");
4    fields = <"example", "_", "_", "_", "_", "_">;
5    Platform.Groups.sharedSpace("", "get", 6, fields, false, "noreply");
6  }
```

6.3.1.4 Terminar uma instância

O término de uma instância ocorre quando todos os tesouros presentes no campo do jogo foram capturados (listagem de código 6.7), ou seja, em termos de implementação, esta verificação ocorre sempre que se verifica uma captura de um tesouro, do lado de cada jogador. Dado que todos os jogadores sabem quantos tesouros existem no campo, basta verificar sempre que é reportada uma captura, se todos os objectos já foram capturados até ao momento. Se já foram, então a instância do jogo é terminada, caso contrário, prossegue-se com o funcionamento da instância.

Listing 6.7: Terminação de uma instância do jogo

```
1  class CaptureEventEndpointListener implements EndpointListener {
2    public void processIncomingMessage(Message msg, EndpointAddress srcAddr,
         EndpointAddress dstAddr) {
3      String treasure = msg.getMessageElement("captured");
4      String team = msg.getMessageElement("team");
5      if(Game.captured+1==Game.treasures) {
6        Game.end();
```

```
7       } else {
8         Game.capture(treasure, team);
9       }
10    }
11  }
12
13  void Game.end() {
14    list = <"end" -> "example">;
15    Platform.Groups.send("example", list, "game", "");
16  }
```

6.3.2 Estados de um jogador

Nesta secção apresentam-se os detalhes da implementação efectuados para cada estado do jogador, bem como as interfaces associadas a cada estado.

6.3.2.1 Registo de um jogador

Como mencionado, um jogador necessita de se registar antes de efectuar qualquer tipo de acção (figura 6.4). Para tal, o jogador necessita de introduzir o nome que o identifica e tentar registar esse nome (listagem de código 6.8).

Caso o jogador consiga registar-se, então este prepara os objectos de escuta necessários: um objecto de escuta para que o jogador saiba quais as instâncias de jogo existentes e um objecto de escuta que contém o tratamento de entrada num jogo.

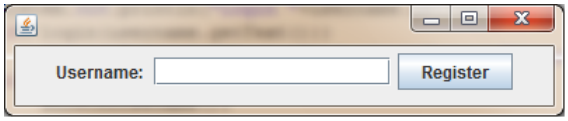

Figura 6.4: Registo de um jogador

Listing 6.8: Registo de um jogador

```
1  success = platform.users.register(username);
2  if(success) {
3    Platform.Groups.addIncomingListener(null, new
         ClientGamesAvailableSharedSpaceListener(this), "sharedspace", "
         gamesavailable"); // Global listener for games available
4    Platform.Groups.addIncomingListener(null, new
         ClientJoinGameSharedSpaceListener(this), "sharedspace", "joingame"); //
         Global listener for joining games
5  }
```

6.3.2.2 Estado inicial

Após efectuado o registo do jogador, este é inserido numa "sala da espera" (figura 6.5). Nesta fase, o jogador consegue ver todas as instâncias de jogos que se encontram em

espera e o jogador consegue entrar numa dessas instâncias ou criar uma nova instância.

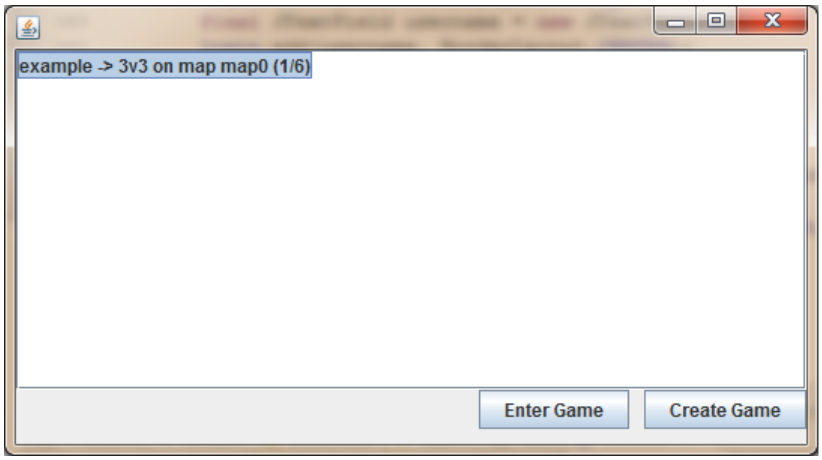

Figura 6.5: Sala de espera

Criação de uma instância

Caso um jogador pretenda criar uma instância do jogo, este deve definir os detalhes de configuração da mesma (figura 6.6).

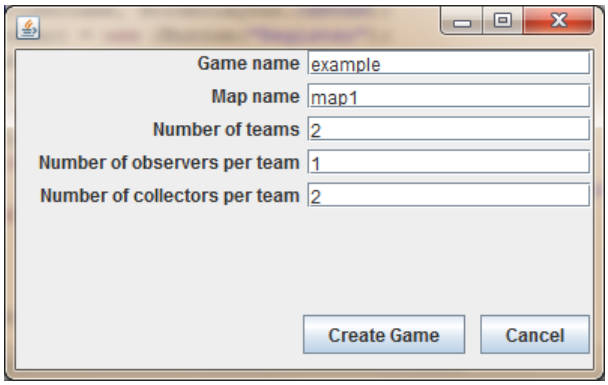

Figura 6.6: Criação de um jogo

Após concluído o processo de criação de uma instância, o jogador que a criou passa ao processo de entrada nessa instância.

Entrada numa instância

Para que um jogador entre numa instância (listagem de código 6.9), este tem de remover o tuplo do espaço partilhado do grupo global referente à instância que pretende entrar. Se conseguir remover com sucesso, reintroduz o tuplo com o número de jogadores activos actualizado (linha 3 e 4) e passa ao processo de junção dessa instância (linha 5); caso não consiga remover, então significa que essa instância já não se encontra no estado de espera e portanto não está a aceitar novos jogadores.

Listing 6.9: Actualização do tuplo relativo ao jogo que se entrou

```
1  class JoinGameListener extends Listener {
2    void processIncomingMessage(msg, src, dest) {
3      fields = <"example", "level\_1", 2, 2, 1, nActive+1>;
4      Platform.Groups.sharedspace("", "update", 6, fields, false, "noreply");
5      Game.join("example");
6    }
7  }
8
9  Platform.Groups.addIncomingListener("", new JoingGameListener(), "sharedspace",
       "joingame");
10 fields = <"example", "_", "_", "_", "_", "_">;
11 Platform.Groups.sharedSpace("", "get", 6, fields, true, "joingame");
```

Após actualizado com sucesso o tuplo que reflecte a instância do jogo ao qual o jogador pretende juntar-se, o jogador necessita de se filiar no grupo global dessa instância "example" (listagem de código 6.10) e registar os objectos de escuta necessários à configuração do jogo.

Listing 6.10: Entrada num jogo definido

```
1  Platform.Groups.join("example");
2
3  Platform.Groups.addIncomingListener("example", new ChatEndpointListener(
       textArea, colors), "chat", ""); // listener for chat before game
4  Platform.Groups.addIncomingListener("example", new
       ClientChangeRoleEndpointListener(this), "role", ""); // Listener for
       changing roles of players
5  Platform.Groups.addIncomingListener(null, new
       ClientReadyStateSharedSpaceListener(this), "sharedspace", "readystate"); //
        Global listener for joining games
6  Platform.Groups.addIncomingListener(null, new ClientRoleSharedSpaceListener(
       this), "sharedspace", "role"); // Global listener for changing roles
7  Platform.Groups.addIncomingListener("example", new
       ClientStartGameEndpointListener(this), "startgame", "");
```

6.3.2.3 Em espera do arranque da instância

Enquanto um jogador está à espera do arranque da instância em que está inserido, este consegue comunicar com todos os jogadores dessa instância, selecionar a sua habilidade e respectiva equipa, bem como colocar-se como preparado (figura 6.7).

Relativamente à comunicação com todos os jogadores dessa instância, cada jogador pode enviar mensagens e o formato das mensagens é apresentado para todos os jogadores da seguinte forma: "USER says: MESSAGE", no qual *USER* representa o nome do jogador e *MESSAGE* contém a mensagem que o jogador enviou.

Quanto à configuração das equipas, utiliza-se o espaço partilhado do grupo do jogo para registar quais os jogadores de cada equipa e respectivas habilidades, existindo um

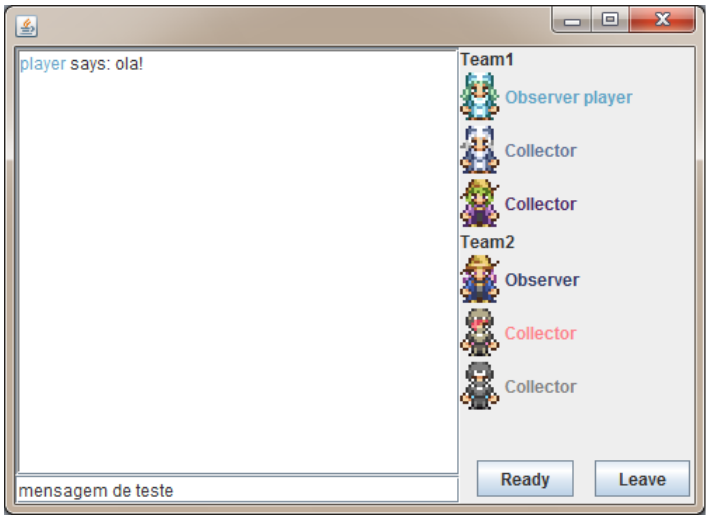

Figura 6.7: Em espera do arranque da instância

tuplo por cada jogador que indica qual a equipa escolhida. Esse tuplo contém três campos: o primeiro campo contém o valor "role" (que corresponde à sua habilidade), o segundo campo contém o nome do jogador e o terceiro campo contém o número que o jogador escolheu neste jogo.

Este terceiro campo serve para identificar univocamente cada jogador na instância, de forma a saber qual a habilidade e equipa a que cada jogador pertence. Essa numeração é sequencial e começa em zero.

Quando um jogador entra numa instância, este tem de consultar a organização que se encontra em vigor no espaço partilhado, de forma a obter a organização correcta das equipas (listagem de código 6.11). A resposta dessa consulta irá actualizar a interface do jogador com a organização dos jogadores. Após obtida essa configuração, o jogador ("player") insere um tuplo que o identifica com a posição -1", que significa que ainda não escolheu uma habilidade.

Listing 6.11: Consulta da organização dos jogadores numa instância

```
1 fields = <"role", "_", "_">;
2 Platform.Groups.sharedSpace("example", "find", 3, fields, true, "role");
3
4 fields = <"role", "player", "-1">;
5 Platform.Groups.sharedSpace("example", "update", 3, fields, false, "noreply");
```

A qualquer momento o jogador pode escolher a sua habilidade e respectiva equipa. Para tal, basta seleccionar na interface o boneco correspondente à sua opção. Para escolher uma habilidade e equipa (listagem de código 6.12), é necessário actualizar o tuplo referente ao jogador, removendo o tuplo (linha 8 e 9) e actualizando esse tuplo após a sua recepção (linha 3 e 4). Para difundir esta alteração pelos jogadores presentes na instância, é gerada uma mensagem para o grupo global da instância "example" (linha 11 e 12).

Listing 6.12: Escolha de uma habilidade e respectiva equipa

```
1  class ClientRoleSharedSpaceListener implements EndpointListener {
2    public void processIncomingMessage(Message msg, EndpointAddress srcAddr,
         EndpointAddress dstAddr) {
3      fields = <"role", "player", "2">;
4      Platform.Groups.sharedSpace("example", "update", 3, toSend, false, "noreply
           ");
5    }
6  }
7
8  fields = <"role", "player", "_">;
9  Platform.Groups.sharedSpace(gameName, "get", 3, fields, true, "role");
10
11 list = <"role" -> 2, "username" -> "player">
12 Platform.Groups.send("example", list, "role", "");
```

Após seleccionada a habilidade e a equipa, o jogador deve indicar que se encontra pronto a começar. Essa indicação consiste em remover o tuplo relativo à lista de jogadores prontos e adicionar o seu nome a essa lista. Sempre que um jogador efectua este processo, verifica se todos os outros jogadores já se encontram prontos; se sim, então é necessário começar esta instância e o jogador que detectou esta situação fica responsável por notificar os restantes jogadores; caso contrário volta-se à configuração do jogo (explicado anteriormente na secção 6.3.1.2).

6.3.2.4 Activo numa instância

A transição para o estado activo, isto é, o arranque da instância consiste em registar os objectos de escuta necessários ao funcionamento dessa instância (listagem de código 6.13) e subscrever o evento *capture* associado ao grupo global da instância. Este evento é utilizado para partilhar todas as capturas efectuadas, ficando ao cargo de cada jogador publicar o evento relativo à captura. Após efectuadas todas as operações necessárias ao funcionamento da plataforma durante o decorrer da instância, a componente gráfica do jogo é lançada e o jogo está pronto a ser jogado.

Listing 6.13: Arranque de uma instância

```
1  Platform.Groups.addIncomingListener("example_team1", new ChatEndpointListener(
       textArea, colors), "chat", ""); // chat with teammates
2  Platform.Groups.addIncomingListener(null, new ChatEndpointListener(textArea,
       colors), "chat", ""); // receive direct messages
3  Platform.Groups.addIncomingListener("example", new GameEndpointListener(this),
       "game", ""); // game listener (movement)
4  Platform.Groups.addIncomingListener(null, new GameSharedSpaceListener(this), "
       sharedspace", ""); // shared space listener (capture treasure)
5
6  Platform.Groups.subscribe("example", "capture");
7  Platform.Groups.addIncomingEventListener("example", captureEventListener, "
       capture", "");
```

A janela que reflecte o jogo consiste numa área de jogo que ocupa a maioria da área visível da janela, sendo o restante espaço ocupado por uma lista de todas as equipas e de quantos tesouros já foram capturados por cada equipa assim como o conjunto de imagens com os jogadores que são da mesma equipa. Estas imagens facilitam, termos de interface, o estabelecimento da comunicação entre jogadores. Por exemplo se um observador desejar comunicar com um dos seus colectores, deve carregar na imagem que representa esse colector. Por fim, existe uma área onde aparece toda a conversação que ocorre no interior da equipa onde o jogador está inserido.

Quanto à comunicação entre os jogadores de uma mesma equipa, esta segue o mesmo formato presente na conversação entre todos os jogadores enquanto estes estão à espera do arranque da instância, isto é, o formato das mensagens é apresentado para todos os jogadores da seguinte forma: "USER says: MESSAGE" , no qual *USER* representa o nome do jogador e *MESSAGE* a mensagem que o jogador enviou.

Visão de um observador

Relativamente aos jogadores que têm a habilidade de observador, estes conseguem ver o campo do jogo por completo (figura 6.8) e conhecem a localização de todos os tesouros presentes no campo do jogo, bem como a localização de todos os jogadores do jogo.

Figura 6.8: Visão do campo do jogo através de um observador

Visão de um colector

Quanto aos jogadores que contam com a habilidade de colector, estes jogadores têm uma visão limitada do campo do jogo (figura 6.9), sendo que só conseguem visualizar a área

do campo por onde já passaram e a área onde se encontram (na implementação do jogo, a distância que os colectores conseguem visualizar à sua volta é de uma unidade).

Dada esta limitação, é importante que os observadores instruam os colectores de forma a que estes capturem os tesouros presentes no campo de forma eficiente.

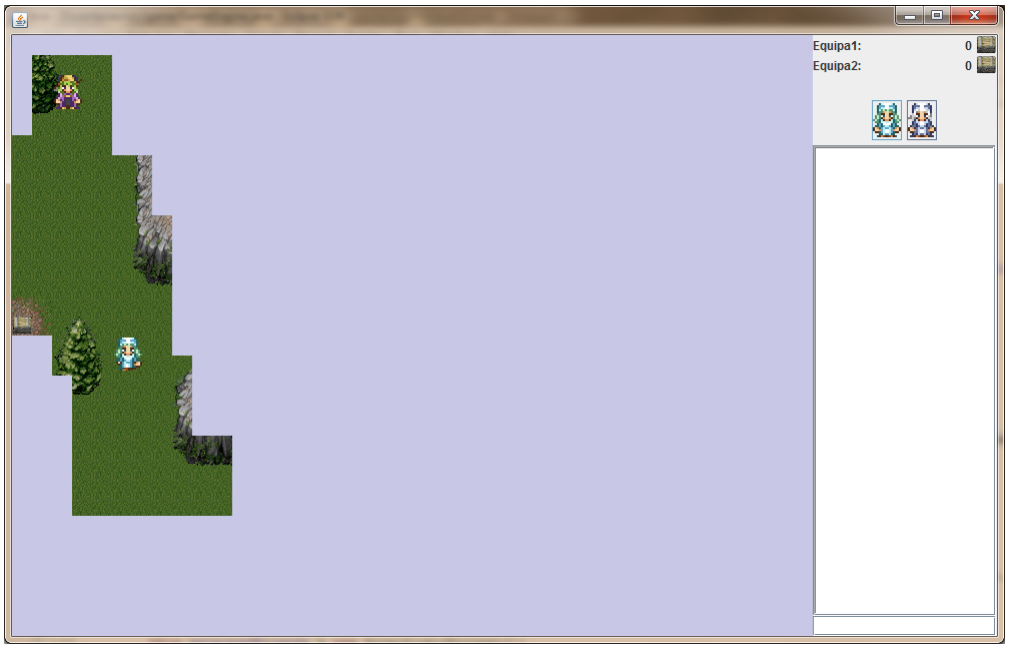

Figura 6.9: Visão do campo do jogo através de um colector

Acções possíveis numa instância

Um jogador numa instância consegue efectuar as seguintes acções: mover-se; capturar tesouros; conversar com jogadores da mesma equipa.

Movimentação

Sempre que um jogador pretende mover-se no campo do jogo, esse movimento tem de ser transmitido aos restantes jogadores. Para tal, é necessário difundir uma mensagem pelos jogadores dessa instância, indicando a nova posição do jogador (listagem de código 6.14).

Listing 6.14: Movimento de um personagem

```
1 list <"name" -> "player", "moveX" -> 5, "moveY" -> 10>;
2 Platform.Groups.send("example", list, "game", "");
```

Captura de um tesouro

A captura de um tesouro ocorre quando um colector se move para a posição onde se encontra um tesouro. A captura com sucesso de um tesouro é verificada pelo incremento

do número de tesouros capturados associado à equipa que capturou esse tesouro.

Listing 6.15: Captura de um tesouro

```
1  class CaptureListener extends Listener {
2    void processIncomingMessage(msg, src, dest) {
3      fields = <"treasure0">;
4      Platform.Groups.sharedSpace("example_team1", "update", 1, fields, false, ""
           );
5      list = <"captured" -> "treasure0"; "team" -> "example_team1">;
6      Platform.Groups.publish("example", list, "capture", "");
7    }
8  }
9
10 Platform.Groups.addIncomingListener("example", new CaptureListener(), "
       sharedspace", "capture");
11 fields = <"treasure0">;
12 Platform.Groups.sharedSpace("example", "get", 1, fields, false, "capture");
```

A captura de tesouros (listagem de código 6.15) consiste em remover do espaço partilhado do grupo do jogo (linha 12), o tuplo referente ao tesouro que se está a capturar. Todos os tesouros presentes no campo estão devidamente identificados e têm um identificador único e respectivo tuplo no espaço partilhado. Se o tuplo relativo a um tesouro não existir no espaço partilhado, significa que esse tesouro já foi capturado por outro jogador.

O jogador que captura um tesouro com sucesso é responsável por colocar o tuplo referente a esse tesouro no espaço partilhado da sua equipa (linha 1 a 8, o tratamento é efectuado dentro do objecto de escuta associado à captura de tesouros), bem como é responsável por gerar uma mensagem associada ao evento "*capture*" , a notificar todos os jogadores do jogo que houve uma captura de um tesouro (linha 6).

Conversação entre jogadores de uma equipa

A qualquer momento um jogador pode comunicar com os restantes jogadores da sua equipa ou com apenas alguns desses jogadores.

A comunicação entre jogadores de uma equipa é relativamente importante, principalmente entre os colectores e os observadores, visto existir a necessidade desta funcionalidade para reportarem a localização de tesouros e guiarem os colectores até aos tesouros.

Listing 6.16: Conversação entre membros de uma equipa

```
1 list = <"msg" -> msg; "user" -> "player">;
2 if(selectedAll())
3   Platform.Groups.send("example_team1", list, "chat", "");
4 else {
5   for(selectedMember as member)
6     Platform.Users.send(member, list, "chat", "");
7 }
```

A comunicação (listagem de código 6.16) consiste no envio de mensagens para todos os membros ou apenas para alguns membros da equipa, sendo essa selecção efectuada de acordo com a descrição apresentada da janela de jogo. Caso se envie uma mensagem para todos os membros da equipa, então é utilizada directamente a primitiva de difusão de mensagens para um grupo (linha 3); caso contrário, procede-se ao envio de uma mensagem directa para cada jogador com que se pretende comunicar (linha 6).

6.3.2.5 Fim de uma instância

Figura 6.10: Equipa vencedora

O fim de uma instância de um jogo ocorre quando todos os tesouros presentes no campo do jogo são capturados (explicado anteriormente na secção 6.3.1.4), sendo apresentado o nome da equipa vencedora (figura 6.10).

Quando o jogador fechar a janela relativa ao jogo, o jogador transita para o estado inicial e é mostrada a sala de espera.

6.4 Conclusão

Este capítulo descreve um jogo colaborativo que serve de exemplo de uma aplicação implementada sobre a plataforma *Imagine*, bem como valida e testa as funcionalidades disponibilizadas pela plataforma *Imagine*.

O capítulo começa com uma listagem das considerações gerais que levaram à elaboração de um jogo colaborativo sobre a plataforma *Imagine*, de seguida especifica o jogo, descrevendo os seus intervenientes e os diferentes estados da instância do jogo e do jogador. Por fim, é descrita a implementação do jogo sobre a plataforma *Imagine*, ilustrando

com pseudo-código aspectos relevantes da implementação que se efectuou.

Existe um protótipo do jogo descrito, tendo a implementação deste sido efectuada à custa do protótipo da plataforma *Imagine* referido no capítulo anterior[1].

[1]Todas as imagens utilizadas no protótipo são de licença LGPL [Pro11], ou seja, são de livre utilização para quaisquer fins sem qualquer tipo de garantia. Todas as imagens foram obtidas no site `http://www.lostgarden.com`.

7

Discussão de Outros Cenários

Neste capítulo são ilustradas possíveis extensões ao jogo apresentado no capítulo anterior. A implementação das extensões pode ser efectuada de forma rápida e simples, visto o jogo ter sido desenvolvido sobre a plataforma *Imagine* e esta permitir a modelação de inúmeros cenários de colaboração.

7.1 Dinamismo nas Equipas

Uma extensão que seria interessante implementar no jogo seria o conceito de dinamismo nas entradas e saídas de equipas ao longo de uma instância do jogo.

Por exemplo, assumindo uma divisão do campo do jogo em quadrantes, seria interessante criar equipas para cada quadrante, permitindo assim efectuar uma sub-divisão das equipas por zona do mapa. Esta divisão é importante quando um campo do jogo é grande e existem diversos observadores e colectores numa mesma equipa, podendo assim restringir a comunicação dessa equipa por quadrantes, de forma a evitar possível confusão gerada pelas diversas indicações por parte dos observadores.

Listing 7.1: Entrada e saída de um grupo

```
1 Platform.Groups.leave("example_quadrant1");
2 Platform.Groups.join("example_quadrant2");
```

A implementação desta extensão passa por gerir as entradas e saídas dos respectivos grupos quando necessário, isto é, quando se detecta que um dado jogador sai de um quadrante e entra num quadrante novo, esse jogador deve actualizar os grupos a que pertence (listagem de código 7.1). Esta implementação assume que os quadrantes são

criados aquando da criação dos grupos necessários na fase de criação da instância do jogo, à semelhança das equipas que são criadas numa instância de jogo.

7.2 Tesouros Dinâmicos

Como mencionado no capítulo anterior, existe a possibilidade de os tesouros serem adicionados dinamicamente com o decorrer do jogo. Este aspecto não se encontra implementado mas seria facilmente concretizado. Para se adicionar um tesouro no campo do jogo é necessário adicionar um novo tuplo no espaço partilhado do grupo do jogo, bem como mencionar a todos os jogadores dessa instância que existe um novo tesouro numa determinada coordenada. Esse tesouro só é mostrado no campo se o jogador tiver visibilidade da coordenada onde o tesouro foi inserido.

A adição de um tesouro pode ser efectuada por um qualquer jogador (listagem de código 7.2). No entanto, idealmente, esta operação deve ser efectuada pelo criador do jogo e esta deve ter um gerador automático que insere o tesouro em coordenadas aleatórias, de forma a não influenciar o jogo para uma determinada equipa.

Listing 7.2: Adicionar um tesouro dinamicamente

```
1 fields = <"treasure1">;
2 Platform.Groups.sharedspace("example", "update", 1, fields, false, "noreply");
3
4 list = <"treasure" -> "treasure1">;
5 Platform.Groups.send("example", list, "new_treasure", "");
```

Outra característica referente aos tesouros que não foi implementada no protótipo é a possibilidade de movimentação dos tesouros. Esta extensão pode ser implementada de diversas formas, como por exemplo, um jogador pode movimentar um tesouro alguns pontos no campo e neste caso o jogador que efectuou essa acção fica responsável por difundir essa alteração (listagem de código 7.3). Outra possibilidade consiste na votação entre os diferentes jogadores de forma a gerar um movimento aleatório para o tesouro, isto é, cada jogador gera um par de coordenadas candidatas à nova posição do objecto e difunde esse par; cada jogador recebe essa informação e efectua a média das coordenadas, movendo o tesouro para essas novas coordenadas.

Listing 7.3: Mover um tesouro

```
1 list = <"treasure1_X" -> "10", "treasure1_Y" -> "10">;
2 Platform.Groups.send("example", list, "move_treasure", "");
```

7.3 Pontuação Global

Uma extensão interessante neste jogo, consiste na criação de uma pontuação global, na qual se consegue ver qual é o número de vitórias alcançadas de todos os jogadores que alguma vez o jogaram.

Tendo em conta que o jogo é totalmente distribuído, normalmente, não existe um ponto central no qual seja possível guardar este tipo de informação. Com a utilização do espaço partilhado, é possível implementar facilmente esta extensão.

Esta funcionalidade consiste em tirar partido do espaço partilhado do grupo global, colocando um tuplo por cada jogador (por exemplo, com o nome do jogador e o número de vitórias desse jogador) e mantendo esses tuplos sempre actualizados (listagem de código 7.4).

Listing 7.4: Actualização da pontuação global de um jogador

```
1 fields = <"player", "10">;
2 Platform.Groups.sharedspace("", "update", 1, fields, false, "noreply");
```

7.4 Novas Habilidades

Esta secção apresenta diversas habilidades que foram tomadas em consideração no desenho do jogo colaborativo.

7.4.1 Teletransporte

Esta habilidade permite que um jogador se teletransporte para qualquer ponto do campo do jogo. Para que os jogadores não abusem desta habilidade, seria necessário adicionar um período de espera entre utilizações.

Uma maneira de implementar esta habilidade consiste em permitir que um jogador possa carregar em qualquer ponto do mapa e teletransportar-se para esse ponto. Seria necessário tratar as colisões de maneira diferente, dado que o jogador poderia escolher um ponto dentro de um obstáculo.

Quanto à implementação desta funcionalidade do lado da plataforma *Imagine*, seria necessário divulgar a nova posição aos jogadores dessa instância (listagem de código 7.5).

Listing 7.5: Actualização da posição de um jogador

```
1 list <"name" -> "player", "moveX" -> 5, "moveY" -> 10>;
2 Platform.Groups.send("example", list, "teletransport", "");
```

7.4.2 Fantasma

Esta habilidade permite que um jogador possa atravessar os obstáculos presentes no campo do jogo. Em conjugação com esta habilidade, seria a possibilidade de adicionar tesouros escondidos dentro de obstáculos, sendo que estes não estariam visíveis aos observadores, obrigando aos jogadores fantasmas a explorar os obstáculos em busca destes tesouros.

Face à implementação desta habilidade, seria necessário ignorar as colisões deste jogador com os obstáculos presentes no campo do jogo e adicionalmente acrescentar tesouros escondidos dentro de obstáculos.

7.4.3 Bloqueante

A opção de bloqueante pode ser dividida em dois aspectos: criação de obstáculos no mapa de forma a bloquear a passagem de todos os jogadores ou movimentação dos obstáculos presentes no campo do jogo.

Na primeira opção, é necessário definir uma interface que permita construir novos obstáculos, por exemplo através da utilização do rato e reportar esse novo obstáculo a todos os jogadores (listagem de código 7.6).

Listing 7.6: Actualização da posição de um obstáculo

```
1 list = <"mountain" -> "1", "X" -> 5, "Y" -> 10, "WIDTH" -> 4, "HEIGHT" -> 4>;
2 Platform.Groups.send("example", list, "new_obstacule", "");
```

Na segunda opção e à semelhança da captura de um tesouro, basta o jogador mover-se em direcção a um obstáculo e esse obstáculo é movido, simulando a acção de "empurrar" por parte do jogador. Para tal, seria necessário detectar a colisão com o obstáculo, movê-lo e reportar essa movimentação aos restantes jogadores (listagem de código 7.7) .

Listing 7.7: Actualização da posição de um obstáculo

```
1 list = <"mountain" -> "1", "X" -> 5, "Y" -> 10>;
2 Platform.Groups.send("example", list, "move_obstacule", "");
```

7.4.4 Assassino

Esta habilidade altera por completo a estratégia do jogo, visto os jogadores não poderem andar livremente no campo do jogo, dado que podem encontrar um assassino de uma equipa adversária. Com a morte de um personagem, esse jogador apenas vê o decorrer do jogo até este terminar.

Para implementar esta funcionalidade é necessário detectar colisões entre jogadores. Se um deles for um assassino, o outro jogador deixa de existir no campo e o jogador que controlava essa personagem fica sem personagem e só pode assistir ao jogo. Esta funcionalidade tem um aspecto interessante: se dois assassinos adversários colidirem é necessário arranjar uma forma de desempate. Esse desempate pode ser efectuado através da remoção de um tuplo do espaço partilhado do grupo de jogo, no qual esse tuplo representa um jogador vivo no campo (listagem de código 7.8).

Listing 7.8: Assassínio de um jogador

```
1 fields = <"player">;
2 Platform.Groups.sharedspace("example", "get", 1, fields, false, "noreply");
```

7.4.5 Bloquear as Outras Equipas Após Captura de Tesouro

Uma outra habilidade interessante neste jogo seria a possibilidade de bloquear os adversários após a captura de um tesouro, dando uma vantagem à equipa que capturou um tesouro.

A implementação desta funcionalidade consiste em bloquear os adversários quando é detectada uma captura de um tesouro por parte de uma equipa adversária, bloqueando qualquer movimentação efectuada pelo jogador. De forma a tornar esta habilidade interessante, a equipa que está bloqueada só poderia ser desbloqueada quando todos os jogadores dessa equipa removessem do espaço partilhado essa indicação (listagem de código 7.9.

Listing 7.9: Actualização da posição de um obstáculo

```
1 // No contexto de cada jogador da equipa que capturou um tesouro
2 fields = <"block">;
3 Platform.Groups.sharedspace("example", "update", 1, fields, false, "noreply");
4
5
6 // No contexto de cada jogador da equipa que ficou bloqueada
7 fields = <"block">;
8 Platform.Groups.sharedspace("example", "get", 1, fields, false, "noreply");
```

7.5 Conclusão

Este capítulo apresenta diversas extensões possíveis de efectuar sobre o jogo desenvolvido. Todas as extensões apresentadas são facilmente e rapidamente implementadas devido às funcionalidades disponibilizadas pela plataforma *Imagine*.

Após apresentadas as extensões possíveis, apresentam-se no capítulo seguinte as conclusões gerais desta dissertação, bem como o trabalho futuro em termo da plataforma e do jogo desenvolvido.

8

Conclusões e Trabalho Futuro

Este capítulo conclui esta dissertação, apresentando um conjunto de considerações finais, uma discussão do trabalho apresentado e trabalho futuro.

8.1 Considerações Finais

Este trabalho teve como principal motivação a aplicação de modelos de grupos em jogos de múltiplos jogadores.

Para tal, numa primeira fase do trabalho, analisaram-se as características fundamentais presentes em diversos jogos de múltiplos jogadores, que se dividiram em três categorias:

- *Organização*: como é que os jogadores se organizam, normalmente através de equipas, em que, geralmente, a filiação às mesmas é dinâmica;
- *Interacção*: ocorre de diversas formas, sendo os mecanismos de comunicação mais comuns baseados em mensagens (ponto-a-ponto ou difusão), baseados em eventos ou baseados no conceito de espaço partilhado;
- *Dinamismo*: através do próprio dinamismo do jogo, isto é, cenários que não contam sempre com o mesmo desenrolar de acção, bem como admitindo a entrada e saída de jogadores ao longo de um jogo.

De seguida, estudou-se um conjunto de plataformas de jogos (*Hero Engine*, *Unity*, *Big World*, *Prime Engine*) que facilitam o desenvolvimento deste tipo de jogos e tentam cobrir todas as características mencionadas. Estas plataformas contam com a mesma característica, em que é dada ênfase à integração do jogador no jogo e toda a interacção entre

os diferentes jogadores é alcançada através do conjunto de servidores disponibilizados para o jogo, não existindo mecanismos explícitos de estruturação e organização entre jogadores.

Tendo em conta essa carência nas plataformas de jogos analisadas, apresentaram-se modelos de grupos que dão ênfase à organização e interacção entre um conjunto de utilizadores.

Os modelos de grupos estudados contam com os seguintes aspectos fundamentais:

- *Gestão de filiação*, que tem como objectivo gerir as entradas e saídas de membros de um grupo;
- *Gestão de comunicação*, que normalmente é efectuada à custa de mensagens, sendo disponibilizados dois mecanismos de envio de mensagens:
  - *Directo*, envio de uma mensagem directa entre dois participantes;
  - *Difusão*, difusão de mensagens por um conjunto de participantes.

Outro modelo de comunicação, também baseado em mensagens, que costuma estar associado aos modelos de grupos, é o modelo de comunicação baseado em eventos, sendo o modelo de publicação/subscrição o mais comum no universo dos modelos de eventos.

Após apresentadas características principais de modelos de grupos, estudaram-se duas plataformas que implementam modelos de grupos: a plataforma *JGroups* e a plataforma *JXTA*.

Em jeito de extensão aos mecanismos de comunicação habituais nos modelos de grupos, surge o modelo de espaço partilhado. Este modelo é desenvolvido e adaptado através de um conjunto de modelos: *GroupLog*, com uma implementação centralizada do conceito, que recorre a uma linguagem de programação em lógica, sendo que o espaço partilhado é alcançado através do modelo *Linda*; *JGroupSpace*, implementação totalmente distribuída utilizando uma linguagem de programação orientada a objectos e *MAGO* que oferece um conjunto de primitivas mais vocacionadas para o desenvolvimento de aplicações multimédia interactivas.

Com base nos modelos de grupos apresentados e nos seus mecanismos de comunicação, desenvolveu-se a plataforma *Imagine*, que tem como principal objectivo facilitar a implementação de aplicações colaborativas, em particular jogos, recorrendo a um modelo de grupos e à utilização do modelo de espaço partilhado, com o intuito de facilitar a partilha de dados e a interacção entre os jogadores.

A modelação dos grupos na plataforma *Imagine* é baseada no modelo *MAGO* e a plataforma eleita para facilitar a elaboração é a plataforma *JXTA*.

As primitivas oferecidas pela plataforma *Imagine* permitem usufruir dos seguintes componentes presentes na plataforma:

- **Gestão de participantes**, responsável pela gestão de todos os participantes e respectiva informação associada a cada participante;
- **Gestão de grupos**, suportando todas as operações permitidas sobre grupos, envolvendo a criação e eliminação de grupos, bem como a gestão da filiação;
- **Comunicação**, dividida em três componentes:
  - **Comunicação directa e difusão**, referente à troca de mensagens directa entre participantes e à difusão de mensagens entre participantes de um mesmo grupo;
  - **Eventos**, cuja concretização se baseia num modelo de publicação/subscrição;
  - **Espaço partilhado**, suportando as funcionalidades que permitem a manipulação de um espaço partilhado de tuplos, interno a cada grupo.

Com a apresentação de todas as primitivas, do ponto de vista de um programador, e ilustrados alguns casos de utilização da plataforma, apresentou-se a implementação que se efectuou no âmbito do trabalho conducente a esta dissertação de forma a obter um protótipo da plataforma.

Em jeito de validação, procedeu-se à construção e implementação de um jogo colaborativo simples, do género caça ao tesouro, que tira partido da maioria das primitivas disponibilizadas pela plataforma *Imagine*. Esse jogo colaborativo conta com um protótipo que foi implementado sobre o protótipo da plataforma.

Por fim, ilustram-se possíveis extensões ao jogo colaborativo proposto e ilustra-se a facilidade de implementação destas sobre a plataforma *Imagine*. Isto deve-se ao facto da plataforma disponibilizar um conjunto de primitivas alargado que permite a implementação de inúmeros cenários colaborativos, com diferentes mecanismos de interacção entre jogadores/utilizadores.

8.2 Discussão

A utilização de modelos de grupos em jogos de múltiplos jogadores permite aliviar a carga de trabalho presente nos servidores que geralmente suportam este tipo de jogos. É possível remover por completo os servidores e desenvolver jogos totalmente distribuídos mas estes requerem que os diversos jogadores tenham recursos que consigam suportar o correr do jogo e a validação de todas as acções efectuadas, dado que este é um trabalho geralmente efectuado por um conjunto de servidores.

Ao incorporar-se um modelo de grupos num jogo, é possível passar algumas tarefas para o lado dos jogadores, como por exemplo, a comunicação entre estes pode ser directa e evitar assim sobrecarregar os servidores.

8.3 Trabalho Futuro

Como trabalho futuro, seria interessante ver as extensões apresentadas no capítulo 7 incluídas num novo protótipo do jogo colaborativo, bem como uma versão final do jogo, de forma a distribuí-lo e testar intensivamente as capacidades da plataforma *Imagine* com um número alargado de jogadores (em simultâneo).

Outra vertente que seria interessante explorar, é a construção de aplicações para dispositivos móveis tendo por base a plataforma *Imagine*. Por exemplo, uma aplicação móvel que permite a um utilizador a partir de um telemóvel participar directamente com jogadores presentes no jogo colaborativo desenvolvido. Neste exemplo, o utilizador da aplicação móvel seria um participante da plataforma e o jogo seria expandido de forma a que os jogadores que se encontram a jogar numa instância, consigam comunicar com jogadores que não se encontram nessa instância, através do envio de mensagens directas entre esses jogadores e participantes.

Bibliografia

[Ban11] Bela Ban. JGroups. http://www.jgroups.org/, Acedido em Janeiro 2011.

[Bar04] Fernanda Barbosa. *Abstrações de Programação Distribuída baseadas em Grupos: o modelo GroupLog*. Tese de Doutoramento, Universidade Nova de Lisboa, 2004.

[Bas95] Jim Basney. *A Distributed Implementation of the C-Linda Programming Language*. Tese de Doutoramento, Oberlin College, Maio 1995.

[BC01] Fernanda Barbosa e José C. Cunha. A coordination language for collective agent-based systems: Grouplog. *Applied Artificial Intelligence: An International Journal*, Janeiro 2001.

[Bio11] BioWare. Star Wars: The Old Republic. `http://swtor.com/`, Acedido em Dezembro 2011.

[Bir05] Kenneth P. Birman. *Reliable Distributed Systems*. Springer Science, Março 2005.

[BJ87] K. Birman e T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21:123–138, Novembro 1987.

[Car87] Nicholas John Carriero, Jr. *Implementation of tuple space machines*. Tese de Doutoramento, New Haven, CT, USA, Dezembro 1987. AAI8809192.

[CDK00] George Coulouris, Jean Dollimore, e Tim Kindberg. *Distributed Systems: Concepts and Design*. International Computer Science Series, Agosto 2000.

[CG89] Nicholas Carriero e David Gelernter. Linda in context. *Commun. ACM*, 32:444–458, Abril 1989.

[Cif11] Frank Cifaldi. World of Warcraft Loses Another 800K Subs In Three Months. `http://www.gamasutra.com/view/news/38460/World_of_Warcraft_Loses_Another_800K_Subs_In_Three_Months.php`, Publicado em Novembro 2011.

[CM96] José C. Cunha e Rui F.P. Marques. Pvm-prolog: A prolog interface to pvm. In *In Proceedings of the 1st Austrian-Hungarian Workshop on Distributed and Parallel Systems, DAPSYS'96*, pág. 173–181, 1996.

[Com93] Compute. Review: Spectre. `http://www.atarimagazines.com/compute/issue157/108_REVIEWS2_SPECTRE.php`, Outubro 1993.

[Cor11] Valve Corporation. Counter-Strike: Source on Steam. `http://store.steampowered.com/css`, Acedido em Junho 2011.

[Cus08] Jorge Filipe Custódio. Jgroupspace - suporte à programação distribuída orientada para grupos. Tese de Mestrado, Universidade Nova de Lisboa - Faculdade de Ciências e Tecnologia, Julho 2008.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, e Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, Junho 2003.

[EG11] Inc. Epic Game. Unreal Tournament. `http://www.unrealtournament.com`, Acedido em Novembro 2011.

[Ent11a] Blizzard Entertainment. World of Warcraft. `http://eu.battle.net/wow/en`, Acedido em Junho 2011.

[Ent11b] Sony Computer Entertainment. PlayStation3 Features Connectivity. `http://us.playstation.com/ps3/features/ps_ps3_connectivity.html`, Acedido em Novembro 2011.

[GBSK02] Darren Govoni, Daniel Brookshier, Juan Carlos Soto, e Navaneeth Krishnan. *JXTA: Java P2P Programming*. Sams Publishing, Março 2002.

[Gel85] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7:80–112, Janeiro 1985.

[Gon01] Li Gong. Jxta: a network programming environment. *Internet Computing, IEEE*, 5(3):88 –95, Maio 2001.

[Hal02] Emir Halepovic. JXTA: A P2P Platform. `http://pages.cpsc.ucalgary.ca/~emirh/pub/JXTA_handout.pdf`, Publico em Outubro 2002.

[Inc11] Deep Silver Inc. Dead Island. `http://www.deadislandgame.com/`, Acedido em Novembro 2011.

[ISI11] The ISIS Project. `http://www.cs.cornell.edu/Info/Projects/ISIS/`, Acedido em Janeiro 2011.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, Julho 1978.

[LH89] Kai Li e Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Novembro 1989.

[Lim11a] BigWorld Pty Limited. BigWorld Technology. `http://www.bigworldtech.com`, Acedido em Setembro 2011.

[Lim11b] BigWorld Pty Limited. BigWorld Technology - BigWorld Server. `http://www.bigworldtech.com/technology/server.php`, Acedido em Setembro 2011.

[LLC11] Id Software LLC. Quake. `http://www.idsoftware.com/games/quake/quake/`, Acedido em Novembro 2011.

[Ltd11] Rovio Entertainment Ltd. Games - Rovio Entertainment Ltd. `http://www.rovio.com/en/our-work/games`, Acedido em Novembro 2011.

[Mak11] Eddie Makuch. Counter-Strike: Global Offensive firing up early 2012. `http://www.gamespot.com/pc/action/counter-strike-global-offensive/news/6328645/counter-strike-global-offensive-firing-up-early-2012`, Publicado em Agosto 2011.

[Mic11] Microsoft. Xbox LIVE Wireless. `http://support.xbox.com/en-US/xbox-live/connecting/wireless`, Acedido em Novembro 2011.

[Mor07] Carmen Pires Morgado. *Um modelo de grupos para aplicações interactivas distribuídas*. Tese de Doutoramento, Universidade Nova de Lisboa - Faculdade de Ciências e Tecnologia, 2007.

[Muh02] Gero Muhl. *Large-Scale Content-Based Publish/Subscribe Systems*. Tese de Doutoramento, Darmastadt University of Technology, Agosto 2002.

[Nob04] David Noblet. JXTA Communications: A Performance Evaluation. `http://davidnoblet.com/assets/Research/Presentations/unh05jxta-perf.pdf`, Publicado em Novembro 2004.

[Ora11] Oracle. JavaSpaces Principles, Patterns and Practice. `http://java.sun.com/docs/books/jini/javaspaces/`, Acedido em Novembro 2011.

[Par11] Laura Parker. GDC 2011 - Clint Hocking: How dynamics create meaning in games. `http://gdc.gamespot.com/story/6301644/how-dynamics-create-meaning-in-games`, Publicado em Março 2011.

[Plc11a] Idea Fabrik Plc. Asynchronous considerations. `http://hewiki.heroengine.com/wiki/Asynchronous_considerations`, Acedido em Setembro 2011.

[Plc11b] Idea Fabrik Plc. HeroEngine. http://www.heroengine.com, Acedido em Janeiro 2011.

[Pri11] Monumental Prime. Monumental Games Groups. http://www.the-prime-engine.com/, Acedido em Setembro 2011.

[Pro11] GNU Project. GNU Lesser General Public License. http://www.gnu.org/licenses/lgpl.html, Acedido em Setembro 2011.

[Rev88] Compute Business Review. TOPS unveils interface to link MS-DOS micros to AppleTalk networks. http://www.cbronline.com/news/tops_unveils_interface_to_link%_ms_dos_micros_to_appletalk_networks_1, Maio 1988.

[RI11] Waldemar Celes Robert Ierusalimschy, Luiz Henrique de Figueiredo. The Programming Language Lua. http://www.lua.org, Acedido em Novembro 2011.

[Sch11] Schooten. JGame: a Java game engine for 2D games. http://www.13thmonkey.org/~boris/jgame, Acedido em Junho 2011.

[Sof11] Take-Two Interactive Software. Sid Meier's Civilization. http://www.civilization.com, Acedido em Novembro 2011.

[Tec11a] Unity Technologies. Unity: Features - Scripting. http://www.unity3d.com/unity/features/scripting, Acedido em Novembro 2011.

[Tec11b] Unity Technologies. Unity: Game Development Tool. http://unity3d.com/, Acedido em Setembro 2011.

[Uni11] Unity Script Reference. http://unity3d.com/support/documentation/ScriptReference/index.html, Acedido em Novembro 2011.

[Wil02] Brendon J. Wilson. *JXTA*. New Riders Publishing, Junho 2002.

[Xam11] Xamarin. Mono - Cross platform, open source .NET development framework. http://www.mono-project.com/, Acedido em Dezembro 2011.

[Zyn11] Zynga. FarmVille - Zynga. http://www.farmville.com/, Acedido em Junho 2011.